Desarrollo de una Estrategia de Migración Automática de JSP a JSF
Development of an Automated Migration Strategy from JSP to JSF

Trabajo de Fin de Máster
Curso 2019–2020

Autor
Santiago Bermúdez Fortes

Director
Manuel Montenegro Montes

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Desarrollo de una Estrategia de Migración Automática de JSP a JSF
Development of an Automated Migration Strategy from JSP to JSF

Trabajo de Fin de Máster en Ingeniería Informática
Departamento de Sistemas Informáticos y Computación

Autor
Santiago Bermúdez Fortes

Director
Manuel Montenegro Montes

Convocatoria: *Junio 2020*
Calificación: *7*

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

15 de julio de 2020

Autorización de difusión

El abajo firmante, matriculado en el Máster en Ingeniería en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Desarrollo de una Estrategia de Migración Automática de JSP a JSF", realizado durante el curso académico 2019-2020 bajo la dirección de Manuel Montenegro Montes en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Santiago Bermúdez Fortes

15 de julio de 2020

Agradecimientos

A mis padres, Miguel y Maricarmen, por darme la vida, su tiempo, amor, cariño y su ejemplo para motivarme a ser una persona de bien. También agradezco a Sebastián y Paloma, mis hermanos menores que me han soportado y apoyado siempre.

A mis tíos Antonio y Mónica, por aceptarme en su casa y apoyarme en esta nueva etapa, y Fran, quien se terminó convirtiendo en un hermano menor.

A Itestra, por la oportunidad y el reto que fue el trabajo, sobre todo la oportunidad de hacer mi TFM con ellos.

Al doctor Manuel Montenegro Montes por su guía durante todo el proyecto.

A mis amigos de la UCM que me han acompañado en esta aventura: Gabriel, Gonzalo, John y Javier.

A los amigos que tengo por el mundo: Andrea, Simone, Peter, Ander, José Eduardo, Eduardo, José Manuel, Alberto, Dudley, Pamela, Martín, Sergio, Brandon, Luis Ernesto, Ricardo, Christian, Iago, Francia, Omaira, Daniela, Estefanía, Melisa, Rodrigo y Tane.

Resumen

Desarrollo de una Estrategia de Migración Automática de JSP a JSF

Al desarrollar una aplicación web es muy conveniente usar un framework, ya que reduce el tiempo para implementar nuevas funcionalidades. Un desarrollo web puede llegar a involucrar muchas tecnologías, y un framework puede ayudar a reducir el conocimiento necesario al facilitar la integración de todas esas tecnologías para que funcionen bien juntas.

JSP (JavaServer Pages) está desarrollado principalmente para crear páginas web dinámicas para pequeñas aplicaciones. Sin embargo, es muy difícil usarlo para aplicaciones a gran escala, ya que JSP es muy flexible, por lo que permite poner lógica correspondiente al modelo de negocio o a un controlador en la vista. Esto provoca que cuanto más crece una aplicación web desarrollada en JSP, más difícil es mantenerla. Por otro lado, JSF (JavaServer Faces) es un sistema basado en componentes que es muy útil para proyectos a gran escala. Este framework utiliza la arquitectura MVC (Modelo, Vista, Controlador) y, por lo tanto, incluso las interfaces de usuario y sus componentes son reutilizables en una página web en particular.

La principal diferencia entre JSP y JSF radica en la forma en que operan. En JSP, un componente tiene un impacto mínimo en otro componente y la comunicación entre componentes depende del programador y no de un framework. Por lo tanto, es posible crear vistas sin tener que escribir el modelo, por lo que la aplicación puede ser más flexible. Pero cuando se trata de aplicaciones web complejas, es difícil mantener la estructura del desarrollo a diferencia de cuando se desarrollan aplicaciones más pequeñas. Puede haber duplicación de código, la estructura puede complicarse y el mantenimiento suele ser difícil. Es por eso que las grandes empresas evitan JSP para aplicaciones grandes ya que podría haber muchos errores aun después de la integración. Por otro lado, JSF se puede dividir en seis fases de desarrollo (restaurar vista, aplicar valores de solicitud, validación del proceso, actualizar los valores del modelo, solicitud de invocación, respuesta de procesamiento) que le dan una estructura rígida al framework. La integración de componentes es perfecta ya que es una tecnología basada en componentes y diseñada para aplicaciones a gran escala.

El código se puede encontrar en: https://github.com/sanbefo/jsp-to-jsf

Palabras clave

Java, JSP, JSF, Framework, Transformación de Código, Análisis Sintáctico, HTML

Abstract

Development of an Automated Migration Strategy from JSP to JSF

In migrating a web development project the best thing to use is a framework, because it reduces the time for implementing new functionalities. A web development project includes many technologies. A framework can help reduce the knowledge needed by making it easier to integrate all those technologies so they work well together.

JSP (JavaServer Pages) is primarily developed to create dynamic web pages for small applications. However, it is very difficult to use it for large-scale applications, since JSP is very flexible, allowing it to put logic corresponding to the business model or a controller in view. This means that the more a web application developed in JSP grows, the more difficult it is to maintain it. On the other hand, JSF (JavaServer Faces) is a component-based system that is very useful for large-scale projects. This framework uses the MVC (Model, View, Controller) architecture and therefore even user interfaces and their components are reusable on a particular web page.

The main difference between JSP and JSF lies in the way they operate. In JSP, one component has minimal impact on the other component. Therefore, it is possible to create views without having to write the model, so the application can be more flexible. But, when it comes to complex web applications, it is difficult to maintain the structure while developing small-scale applications. There can be code duplication, the structure can become a mess and maintenance can be difficult. That is why big companies avoid JSP for large-scale applications and there could be a lot of bugs and errors after integration. On the other hand, JSF can be divided into six phases of development (Restore View, Apply Request Values, Process Validation, Update Model Values, Invoke Application, Render Response) that allow the framework to grow smoothly. The integration of components is seamless as it is a component-based technology and designed for large-scale applications.

The code can be found in: https://github.com/sanbefo/jsp-to-jsf

Keywords

Java, JSP, JSF, Framework, Transformation, Parser, HTML

Table of Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

"The Legacy Problem is like global warming. Some people don't believe there's a problem at all but the ones who do know that it won't go away by itself, it will only get worse"
— Matjaz Jug, September 2008

1.1. Motivation

The motivation for this Thesis is to create a tool that can reduce the time invested in migrating projects from JSP (JavaServer Pages) to JSF (JavaServer Faces).

In migrating a web development project the best thing to use is a framework, because it reduces the time for implementing new functionalities. A web development project includes many technologies, such as HTML, CSS, JavaScript, a backend language, a database technology and many more inbetween. A framework can help reduce the knowledge needed by making it easier to integrate all those technologies so they work well together. frameworks also include security features by default that greatly reduce the cost of implementing features. They have guidelines and a structure that, if followed, it becomes easier to organize the project. They also do a great job at including other libraries and making them work together [1].

JSP is primarily developed for creating dynamic web pages for small applications. However, it is very difficult to use it for large-scale applications as they are developed with a certain framework and component-based system. JSF is a component-based system that is highly useful for large-scale projects. JSF uses MVC (Model, View, Controller) and hence, even the user interfaces and its components are reusable in a particular web page.

JSF is a proper framework and that is why it is widely used in the web development industry. It uses XML for view templates. FacesServlets are responsible for processing requests and sending required view templates, creating component trees, processing events and rendering response to the clients. The state of the components is saved and later is retrieved before creating another view. On the other hand, JSP is a request-driven technology and it is translated into servlets at runtime. Even though it is request-driven, instead of using it independently, it can be used with view components of any server-side MVC design.

The main difference between JSP and JSF lies in the way they operate. In JSP, one component has minimal impact on the other component. Therefore, it is possible to create views without having to write the model, so the application can be more flexible. But, when it comes to complex web applications, it is difficult to maintain the structure while

developing small-scale applications. There can be code duplication, the structure can become a mess and maintenance can be difficult. Since JSP is a core technology, there can be different developers for different components but care must be taken when it comes to integration as it can go wrong. That is why big companies avoid JSP for large-scale applications and there could be a lot of bugs and errors after integration. On the other hand, JSF can be divided into a fixed sequence of development phases. The integration of components is seamless as it is a component-based technology and designed for large-scale applications. It is possible to create and restore views that are used for showing information to the clients, update values as per requirements without major changes, process validations and data type conversions. Similarly, it is possible to invoke applications to fulfill any request and render responses [2].

There are some challenges to migrate a system, here are some that must be taken into account [3]:

- The new system cannot be a copy of the old one; it always has new requirements in line with the business.
- New technology must be used to support current/future business needs.
- Knowledge needed to migrate the software may not be documented, so it slows down the developing process.
- Business continuity must be ensured at all times during the migration.
- Data migration must be eficient and data consistency must be ensured.

1.2. Objectives

The Thesis objective is to create a software tool capable of migrating a web view file automatically from JSP to JSF, by parsing the files and transforming their tags and code into a JSF valid syntax, in order to accelerate the migration of old projects into more modern ones.

1.3. Workplan

- Read about JSP and JSF to familiarize with the technology.
- Divide what tags are easy/simple to change and what tags are more complicated to start transforming incrementally.
- Test with simple examples that contain HTML tags that must be transformed into a JSF allowed syntax.
- Report the results of the implementation.

1.4. Document Structure

The structure of the document is as follows:

- In chapter 2, the state of the art is described: what is HTML, best practices for it, parsers, background for JSP and JSF and migrating projects.

- In chapter 3, we describe the proposed solution and implementation, we mention the used packages and explain some setbacks we had in the development.
- In chapter 4, we explain how to run the program and describe the tests and results.
- In chapter 5, conclusions, lessons learned and future work are presented.

Chapter 2

State of the Art

Summary: In this chapter the state of the art of the technologies involved in the project is describred. This includes HTML, Web development, Parsers, Java, JSP, JSF and migrating projects.

2.1. HTML Background

HTML stands for Hypertext Markup *Language*, so, technically, HTML is a programming language, but HTML and CSS are declarative languages, in contrast to other languages such as Java, C, Python, or Javascript, which explicitly express how a computation is done. HTML and CSS do not specify a computation, but the structure of a document [4].

This language was created by *Sir Tim Berners-Lee* in late 1991. There have been several versions of it throughout history [5; 6]:

- HTML 1.0

  It was released in 1993 with the intention of sharing information that can be readable and accessible via web browsers. It had very limited features which greatly limited what a developer could do to design web pages.

- HTML 2.0

  It was published in 1995, and it contained all HTML 1.0 features along with a few more, which remained as the standard markup language for designing and creating websites until January 1997. Netscape, the leading browser at that time, introduced new tags and attributes called the *Netscape Extension Tags*. Other browsers tried to duplicate them but Netscape did not specify their new tags and so these extension tags did not work in other browsers. It led to considerable confusion and problems when web developers used these tags and attributes and then saw that they did not work as expected in other browsers.

- HTML 3.0

  A group of people working under *Dave Raggett*'s leadership introduced the HTML 3.0 draft which included many new enhancements to HTML. However, most browsers only implemented a few. Another reason why HTML 3.0 did not make it was because it was considered too big and it slowed browsers down. Future versions were to be

introduced in a more "modular" way so that browsers could implement them little by little.

- HTML 3.2 (WILBUR)

  As more browser-specific tags were introduced, it became obvious that a new standard was needed. For this reason, the Word Wide Web Consortium (W3C), founded in 1994 to develop common standards for the evolution of the World Wide Web, drafted the WILBUR standard, later known as HTML 3.2. It became the official standard in January, 1997.

- HTML 4.0 (COUGAR)

  It introduced new functionality, most of which came from the expired HTML 3.0 draft. This version was recommended in December, 1997 and used as a standard in April, 1998.

- XHTML

  It stands for *EXtensible HyperText Markup Language*. It does not bring along new tags. The purpose of XHTML is to address new browser technologies. Nowadays, web pages are viewed in browsers through cell phones, cars, televisions, etc. and in many cases, these devices do not have the computing power of desktop or notebook computers and so are not able to accommodate poor or sloppy coding practices. XHTML is designed to address these issues. It also addresses the need for people with disabilities (such as the blind and visually impaired) to access the Internet. Thus web pages written in XHTML allow them to be viewed on a wide range of browsers and internet platforms.

  XHTML is the result of the W3C's work to bring a standard to provide rich high quality web pages through these varied devices. XHTML became an official W3C recommendation in January, 2000. XHTML2 was cancelled in 2009, having HTML5 take its place.

- HTML5

  It is the new web standard. It follows HTML 4 and XHTML. The purpose of HTML5 is to provide two things:

  - Improve the language.
  - Support the latest multimedia technologies.

  To accomplish this, it reduces the need for external plug-ins (such as Flash plug-ins), and more markup elements (tags) to replace scripting. HTML5 is also device independent (understood by computers and many devices today) while also keeping it easily readable by humans.

2.1.1. HTML Best Practices

HTML can be read by the browser without showing any errors because of its flexibility, but this is not necessarily an advantage because the end result can be different from what was expected, and also this makes it hard to maintain. That is why there are some best practices that can be found here [7; 8]:

- Always Declare a Doctype.

  The doctype declaration should be the first element in HTML documents. The doctype declaration tells the browser about the XHTML standards used and helps it read and render a webpage correctly.

- Use Meaningful Title Tags.

  The `<title>` tag allows one to make a web page more meaningful and *search-engine friendly*. For example, the text inside the `<title>` tag appears in Google's search engine results page, as well as in the user's web browser bar and tabs. Figure 2.1 is a good example.

- Use Descriptive Meta Tags

  Meta tags make the web page more meaningful for user agents like search engine spiders. The description meta attribute describes the basic purpose of your web page (a summary of what the web page contains). Again, look at figure 2.1.

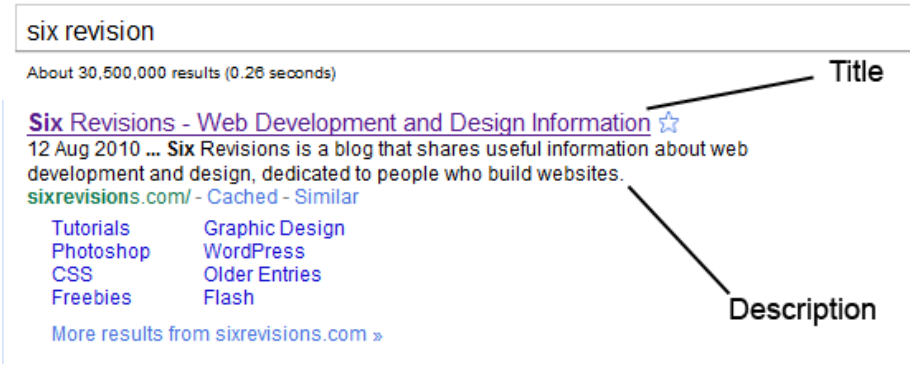

Figure 2.1: `title` and `meta` example (Image found in [8])

- Use the Right HTML Element at the Right Place

  For example, use `<em>` for emphasis and `<strong>` for heavy emphasis, instead of `<i>` or `<b>` (which are deprecated).

- Close Tags

  The W3C specifies that all tags should be closed. Some browsers may still render pages correctly, but not closing tags is invalid under standards.

- Use Lower Case Markup

  It is an industry-standard practice to keep the markup lower-cased. Capitalizing the markup will work and will probably not affect how web pages are rendered, but it does affect code readability.

- Write Consistently Formatted Code

  A cleanly written and well-indented code base shows professionalism, as well as consideration for other people that might need to work on that code.

  Writing properly indented clean markup from the start will increase code's readability.

- Do not use inline styles and scripts

  The document will quickly become cluttered and unreadable otherwise. It is better to use external stylesheets. Same goes for inline JavaScript as for inline CSS. Apart from readability issues, this will make the document heavier and harder to maintain.

- Only use critical inline CSS

  Critical CSS must be placed at the top of the web page. By doing so, users will get to see the first portion of the page rendered more quickly. Critical CSS refers to the minimum CSS that is required to render the top of the page that a user sees first when landing on the site. The order of the link tags can affect how CSS rules are applied so they must be placed carefully. If there are separate files for resets or 3rd party libraries, they must be placed first and then the rest.

- Script tags must be placed at the bottom

  Officially, script tags live inside the head, but if they are placed at the bottom of the document, before the closing tag of the body, their download can be delayed and allow the document to load first in the DOM, show it to the user and then request the scripts. This is because the browser interprets the document from top to bottom, line by line. When it gets to the head and comes across a script tag, it starts a request to the server to get the file. If the file is very big, it will keep loading and the user will only see a blank page because it is still loading the head, so it should be moved to the bottom. This way, all the content of the body will get loaded in before it loads the content of the script tag. Figure 2.2 is a good example of how the scripts affect the DOM loading.

- Use `alt` tags for images

  The `alt` tag specifies an alternate text for the image, so in case it cannot be displayed for any reason, this text will be shown instead. Search engines prioritize websites that contain `alt` tags for images so they rank higher in search results.

- Use one `h1` per page

  The most important text should be placed here, which describes the content of the page, Using multiple `h1` tags per page is not a good idea and not advised, because it can affect the search engine results. This aids search engines at indexing the site the right way.

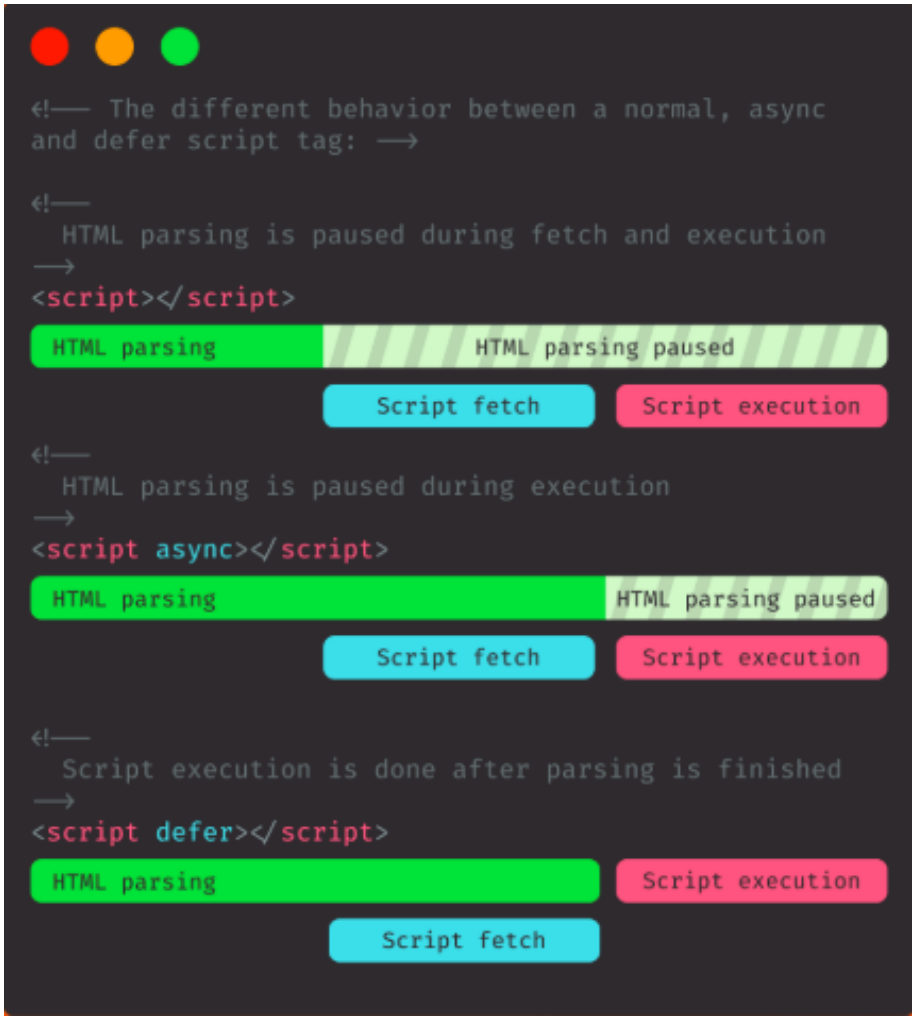

Figure 2.2: Loading scripts DOM speed comparison (Image found in [7])

2.2. HTTP

HTTP stands for *Hypertext Transfer Protocol*. It is the foundation of the World Wide Web. It defines how messages are formatted and transmitted, and what actions Web servers and browsers should take in response to various commands. A server is listening for a request and sends back a response, then the communication terminates. When the user enters a URL in a browser, this actually sends an HTTP command to the Web server directing it to fetch and transmit the requested Web page. Figure 2.3 ilustrates the HTTP cycle.

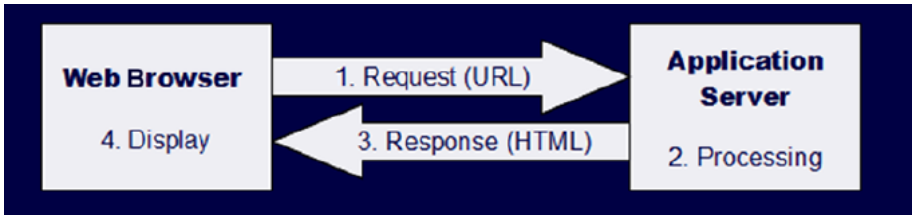

Figure 2.3: HTTP request-response cycle (Image found in [9])

Any request acts independently of the previous ones. To proceed with a user session in the application, a session management needs to be implemented on top of HTTP. The goal of this session management is to remember the current state of the application. The server transfers the complete state information to the client. This approach keeps memory consumption on the server lean. Sending too much information over the web has its drawbacks because latency slows applications down. Another drawback is security, because if it is not secure enough, the information sent might be read by unauthorized persons [9].

2.2.1. HTTP Errors

HTTP Status Codes are Error Messages that the server shows in order to tell either the user or the developer that there is something wrong [10].

- 400 Bad File Request

  Usually means the syntax used in the URL is incorrect.

- 401 Unauthorized

  Server is looking for some encryption key from the client and is not getting it. Also, wrong password may have been entered.

- 403 Forbidden/Access Denied

  Similar to 401; special permission needed to access the site – a password and/or username if it is a registration issue.

- 404 File Not Found

  Server cannot find the requested file/information.

- 408 Request Timeout

  Client stopped the request before the server finished retrieving it.

- 500 Internal Error

  Could not retrieve the HTML document because of server-configuration problems.

- 502 Service Temporarily Overloaded

  Server congestion, too many connections, high traffic.

- 503 Service Unavailable

  Server busy, site may have moved, or the internet connection has been lost.

2.2.2. HTTP Verbs

HTTP defines a set of request methods to indicate an action to be performed by the server. Here is a list of the most common ones [11].

- GET

  It requests a representation of the specified resource.

- POST

  It is used to submit an entity to the specified resource, causing a change in state of the server.

- DELETE

  It deletes the specified resource.

- PATCH

  It is used to apply partial modifications to a resource.

2.2.3. HTTPS

The name HTTPS stands for *Hypertext transfer protocol secure*. HTTPS is encrypted in order to increase security of data transfer. This is particularly important when users transmit sensitive data, such as logging into a bank account, email service, or health insurance provider. Any website, especially those that require login credentials, must use HTTPS.

HTTPS uses an encryption protocol to encrypt communications. The protocol is called *Transport Layer Security* (TLS). This protocol secures communications by using an asymmetric public key infrastructure. This type of security system uses two different keys to encrypt communications between two parties, as shown in figure 2.4 [12].

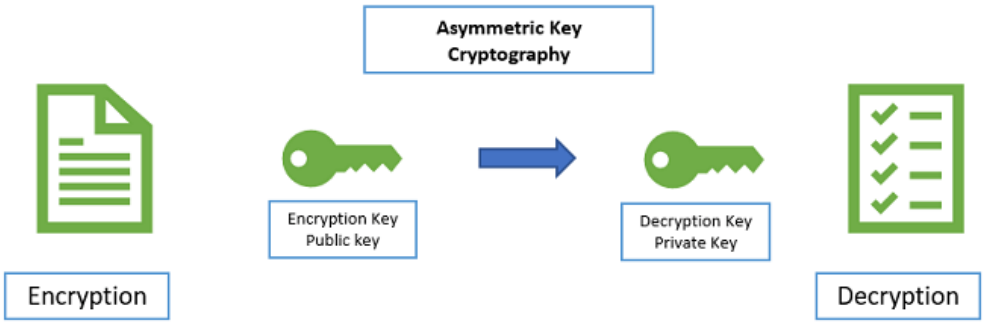

Figure 2.4: Asymmetric Key Encription (Image found in [13])

- The Private key: this key is controlled by the owner of a website and it is kept, as the reader may have speculated, private. This key lives on a web server and is used to decrypt information encrypted by the public key.
- The Public key: this key is available to everyone who wants to interact with the server in a secure way. Information that is encrypted by the public key can only be decrypted by the private key.

2.3. Web Development

> *A web application is a client-server application interacting dynamically with the user via a web browser.*
>
> Michael Müller, [9]

We can define the web as a means of sharing information, documents and resources between users via The Internet. The early Web consisted on a collection of texts formatted

in HTML hosted on servers. The Web itself appeared in the early 1989. It was firstly designed to meet the request for information-sharing among universities and scientific institutions around the world [14]. Figure 2.5 shows the differences betweeen Web 1.0, Web 2.0 and Web 3.0.

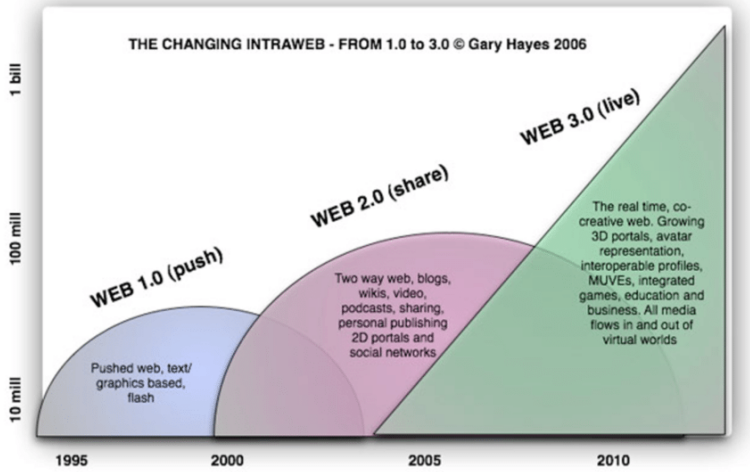

Figure 2.5: History of Web (Image found in [14])

Before The Web was created, information was distributed in several documents spread around the world and it was difficult to access it.

2.3.1. Web 1.0

Web 1.0 represents the basics of the web. It was used mostly until 2003, and it was just a readable site with raw data of the World Wide Web. The user can only Search and read the information through the browser, but cannot share on the site. It is made of static (fixed) information. Technologies used in Web 1.0 are HTML, HTTP, URI, XML, XHTML and CSS. Web 1.0 is very slow and the user needs to refresh the site every time when new information is added to the site. This is because it just works in one direction. The thing it solves is the access to information, because there are links between websites that makes resources quicker and easier to find. Figure 2.6 represents how this looks.

A web application is a client-software application run by the user in a browser. The main function of a browser is to show the information received from a server and send the user's data back. The main advantage of this approach is the fact that clients do not depend on the user's operating system; therefore, web applications are cross-platform services. Due to this universal feature, web apps became very popular in 1990s and 2000s.

2.3.2. Web 2.0

In 1995 Netscape Communications presented JavaScript, a client-side scripting language that enables programmers to improve the user interface with dynamic elements. JavaScript made the Internet faster and more productive because the data was no longer sent to the server to generate the whole web page. The Embedded scripts fulfil various tasks on the specific downloaded page “right on the spot”. JavaScript is one of the three most notable technologies (with HTML and CSS) of content production for WWW. It has an Application Programming Interface (API) that enables developers to work with texts,

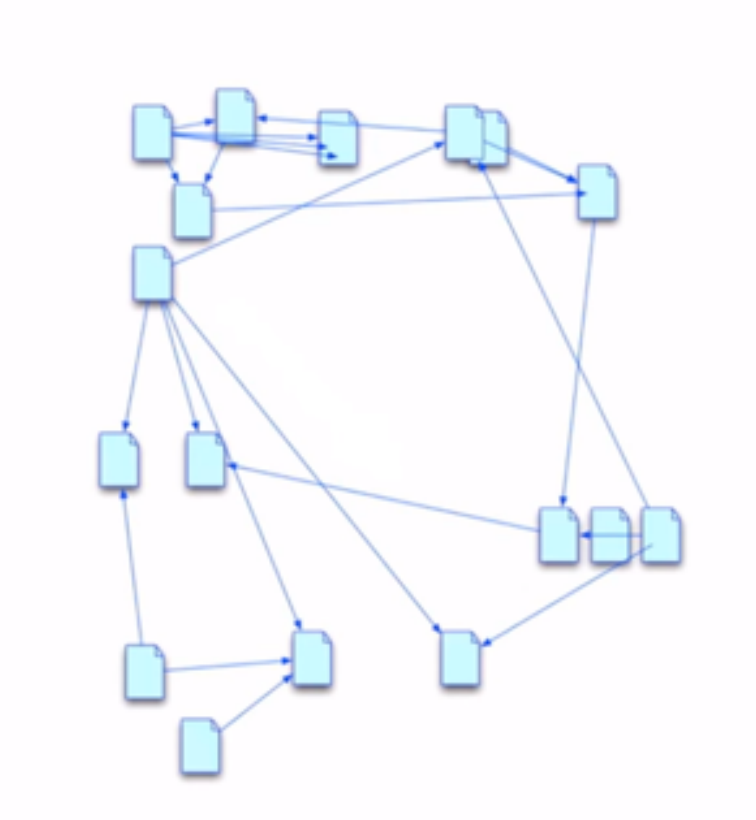

Figure 2.6: Web 1.0 (Image found in [15])

dates and various regular expressions. In fact, it does not have input/output that makes the machine "communicate" with the outside world.

In 1996 Macromedia Flash was introduced. It was a revolutionary innovation that made the Web "brighter" and interactive. This vector-based animation player enabled developers to enrich web pages with animation. This multimedia software platform works with animations, different types of browser games, vector graphics, Internet and mobile applications. The majority of the Internet before 2000 was full of websites that used embedded interactive multimedia content on their pages. Animated ads could be seen and videos that overloaded webpages with colors and unnecessary "movement". Very soon, the popularity of Flash declined. Webpages gained their regular look. The user's work was no longer interrupted by the odd and unexpected ads and streaming videos as much as they slowed the work of the website and consumed the additional traffic [16].

2.3.2.1. AJAX

AJAX stands for Asynchronous JavaScript and XML, and it represents a fundamental shift in what is possible on the Web. AJAX is not a technology, it is a combination of several technologies. It enabled developers to create asynchronous web apps. It made it possible for users to work on the Web faster and better because there is no need to download the whole page. AJAX incorporates:

- Standards-based presentation using XHTML and CSS.
- Dynamic display and interaction using the Document Object Model.

- Data interchange and manipulation using XML and XSLT (Extensible Stylesheet Language Transformations).
- Asynchronous data retrieval using XMLHttpRequest.
- JavaScript binding everything together.

Without AJAX, the classic web application model works like this: Most user actions in the interface trigger an HTTP request back to a web server. The server does some processing — retrieving data, crunching numbers — and then returns an HTML page to the client.

An AJAX application eliminates the start-stop-start-stop nature of interaction on the Web by introducing an intermediary — an AJAX engine — between the user and the server. It is like adding a layer to the application. Instead of loading a webpage at the start of the session, the browser loads an AJAX engine written in JavaScript. This engine is responsible for both rendering the interface the user sees and communicating with the server on the user's behalf. The AJAX engine allows the user's interaction with the application to happen asynchronously — independent of communication with the server. So the user is never staring at a blank browser window waiting for the server to deliver the results. Figures 2.7, 2.8 and 2.9 show the differences.

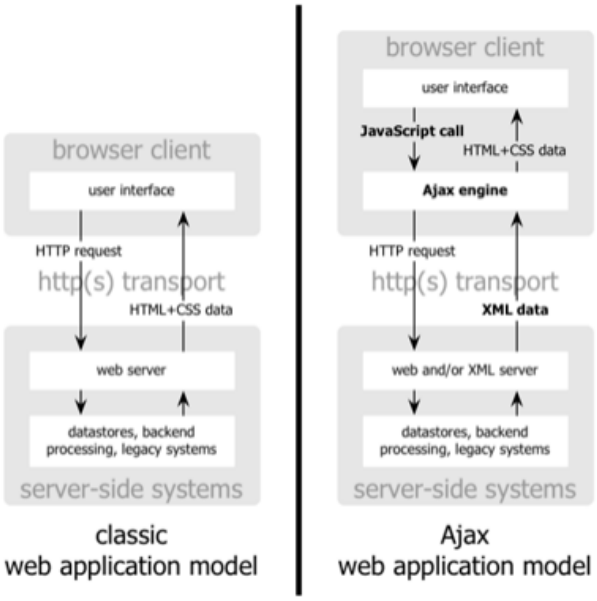

Figure 2.7: The traditional model for web applications (left) compared to the AJAX model (right) (Image found in [17])

Every user action that generates an HTTP request takes the form of a JavaScript call to the AJAX engine instead. Any response to a user action that does not involve the server — such as simple data validation, editing data in memory, and even some navigation — the engine handles it on its own. If the engine needs something from the server in order to respond — if it is submitting data for processing, loading additional interface code, or retrieving new data — the engine makes those requests asynchronously, using XML, without stalling a user's interaction with the application.

In 2004, Web 2.0 was presented formally by *Dale Dougherty*, who was vice-president of O'Reilly Media. It is also called the *read and write web* (writable), it represents a new

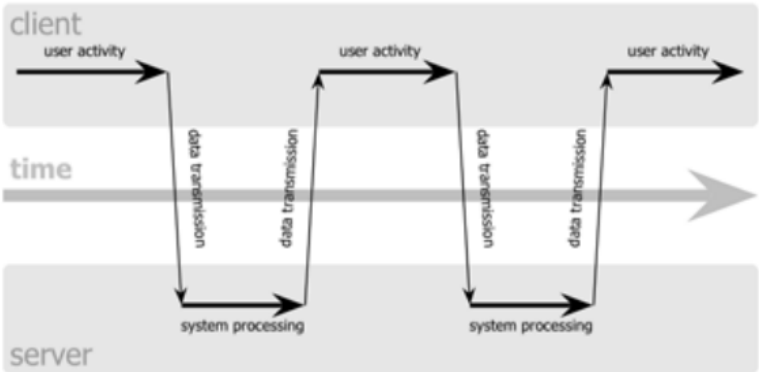

Figure 2.8: Classic web application model (Synchronous) (Image found in [17])

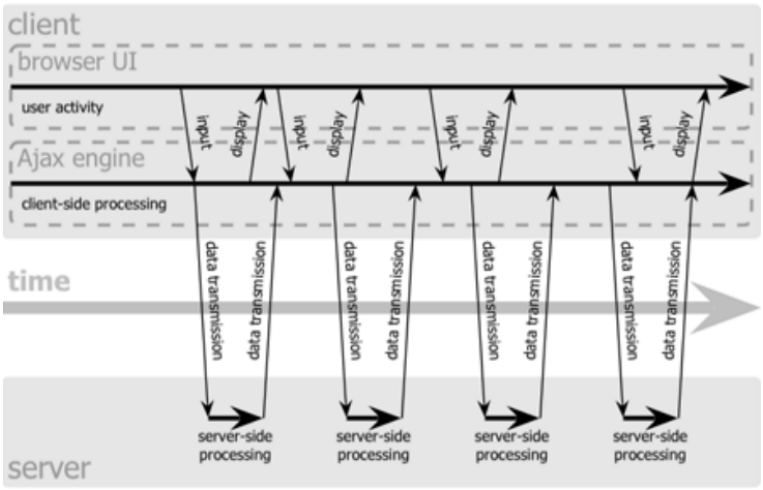

Figure 2.9: AJAX web application model (Asynchronous) (Image found in [17])

method to use the current internet technologies, and the web could become bi-directional. Actually the Web 1.0 presents the user the possibility to upload and download from the webpage like a provider (site admin) but with limited control. The users of Web 2.0 have more interaction with less control. Technology infrastructure of Web 2.0 consist of some rules such as AJAX technology in internet such as JavaScript and XML, DOM, REST, XML and CSS. The Web 2.0 allows users the ability to create social activities and communicate with each other. But these properties also consider issues because the user can be hacked in privacy and personal information security.

The problem with Web 2.0 is that web applications do not share information, so apps that should have the same data are hard to keep up to date. This is called Application Silos (See figure 2.10).

2.3.3. Web 3.0

The third and current version of the web started in 2014 known as executable web. It allows users the ability to interact with dynamic applications. Web 3.0 implies to convert the web into a huge database [17].

The Semantic Web is an extension of the World Wide Web through standards set by the W3C. Its goal is to make Internet data machine-readable.

According to the W3C, "The Semantic Web provides a common framework that allows

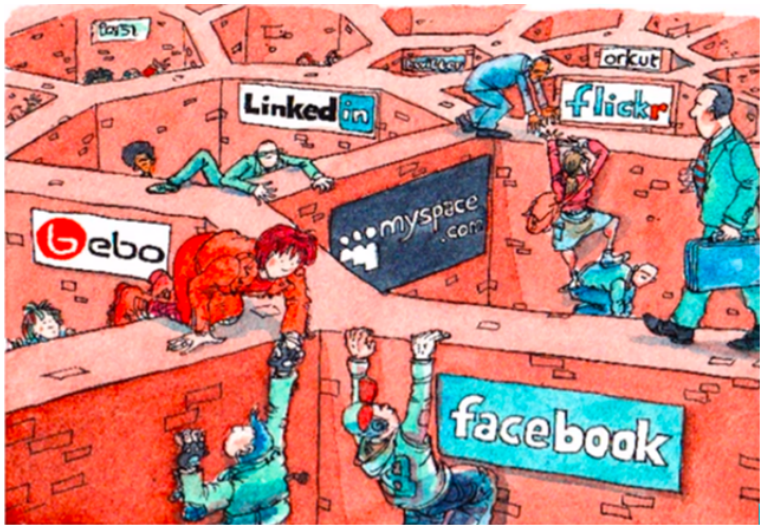

Figure 2.10: Application Silos [15]

data to be shared and reused across application, enterprise, and community boundaries". The Semantic Web is therefore regarded as an integrator across different content and information applications and systems. Instead of having URLs bewteen documents, there are URLs between *facts* (data). If that information is ever updated, it can be linked from a source to any other source and to be understood by computers so that they can perform increasingly sophisticated tasks on the user's behalf. Figure 2.1 is a good example of this.

The term Web 3.0 was coined by *Tim Berners-Lee* for a web of data that can be processed by machines, that is, a web in which much of the content is machine-readable. Berners-Lee originally expressed his vision of the Semantic Web in 1999 as follows: "I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A *Semantic Web*, which makes this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines" [18].

The Semantic Web abstracts away the document and application layers involved in the exchange of information.

2.3.4. Progressive Web Apps

Native desktop applications are known for being incredibly rich and reliable. They can read and write files from the local file system, access hardware connected via USB, serial or bluetooth, and even interact with data stored on your device, like contacts or calendar events. In native applications, it is possible to do things like take pictures, see playing songs on the home screen, or control song playback while in another app. Native applications feel like part of the device they run on.

In terms of capabilities and reach, native apps represent the best of capabilities while web apps represent the best of reach. Progressive Web Apps (PWA) are built and enhanced with modern APIs to deliver native-like capabilities, reliability, and installability while reaching anyone, anywhere, on any device with a single codebase. Progressive Web Apps are web applications that have been designed so they are capable, reliable, and installable. These three pillars transform them into an experience that feels like a native application [19]:

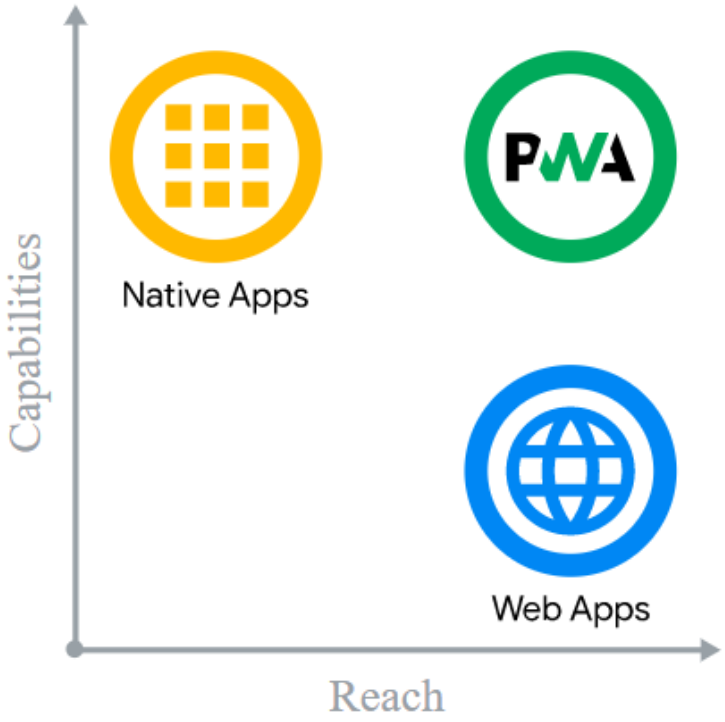

Figure 2.11: Capabilities vs. reach of native apps, web app, and progressive web apps

- Capable

  While some capabilities are still out of the web's reach, new and upcoming APIs are looking to change that, expanding what the web can do with features like file system access, media controls, app badging, and full clipboard support.

  Progressive Web Apps must also be fully integrated with the services that users usually expect when navigating either a web app or a mobile app, like mobile payments, access to the user's location or push notifications.

- Reliable

  A reliable Progressive Web App feels fast regardless of the network. Speed is critical for a good user experience; it helps avoid the bounce rate of the site. Performance affects the entire experience, from how users perceive the application to how it actually performs. Users expect apps to start up on slow network connections or even if they are offline. When a request is not possible, they expect to be told there is trouble instead of silently failing or crashing, just like Google Chrome's downasaur.

  Security is very important as well, the conection must be secure between the site and the user.

- Installable

  Installed Progressive Web Apps run in a standalone window instead of a browser tab. It is possible to search for them on a device and jump between them with the app switcher, making them feel like part of the device they are installed on. New capabilities open up after a web app is installed. Progressive Web Apps should be able to register to accept content from other applications.

There are three ways to approach a Progressive Web App [20]:

- Building it from the ground up. A redesign is a good opportunity to take on this approach.
- Building a simpler version of the product (light or mobile).
- Building just a simple feature that can have a high impact on the usage of the product.

2.4. Parsers

A parser is a compiler or interpreter component that breaks a program's source code into smaller elements. It takes the input in the form of a string and builds a data structure in the form of an abstract syntax tree.

It is commonly used as a component of an interpreter or a compiler. The overall process of parsing involves three stages:

- Lexical Analysis

  It is used to produce tokens from a stream of input string characters, which are broken into small components to form meaningful expressions.

- Syntactic Analysis

  Checks whether the generated tokens form a meaningful expression. This uses a context-free grammar that defines algorithmic procedures for components. These work to form an expression and define the particular order in which tokens must be placed.

- Semantic Parsing

  The final parsing stage is where the meaning and implications of the validated expression(s) are determined.

The main purpose of a parser is to determine if the input data may be derived from the start symbol of the grammar. If yes, input data can be derived as figure 2.12 shows:

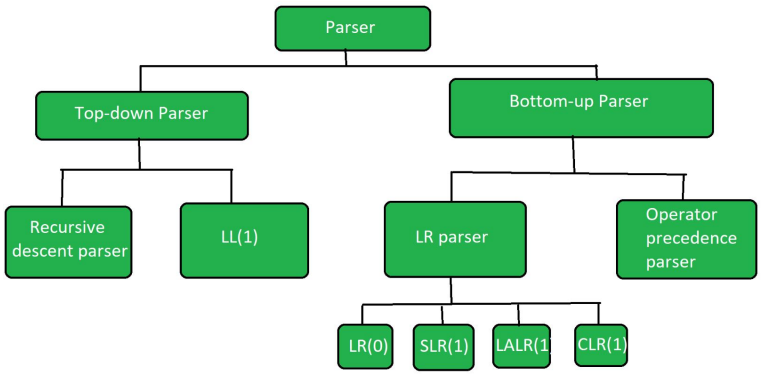

Figure 2.12: Types of Parsers (Image found in [21])

- Top-Down Parser

  It starts from the start symbol and ends on the terminals. It uses leftmost derivation. There are 2 types of parsers, and they can be classified into Recursive descent parsers and Non-recursive descent parsers.

  - Recursive descent parsers:
    It generates the parse tree by using recursion.
  - Non-recursive descent parsers:
    It uses parsing table to generate the parse tree instead of backtracking.

- Bottom-Up Parser

  It starts from non-terminals and ends on the starting symbol. It uses the reverse of the right most derivation. Further Bottom-up parser is classified into 2 types: LR parser, and Operator precedence parser.

  - LR parser:
    Is the bottom-up parser that generates the parse tree for the given string by using unambiguous grammar. It follow reverse of right most derivation. There are many types of LR parsers, among them: LR(0), SLR(1), LALR(1), CLR(1).
  - Operator precedence parser:
    It generates the parse tree from a given string but the only condition is that two consecutive non-terminals never appear in the right-hand side.

Parsers are widely used in the following technologies [22]:

- Java and other programming languages.
- HTML and XML.
- Database languages, such as SQL.
- Scripting languages.
- Protocols, such as HTTP and Internet remote function calls.

2.5. Java [23]

Java was originally developed by *James Gosling* at Sun Microsystems (now Oracle) and released in 1995 as a core component of Sun Microsystems' Java platform. Java is a general-purpose programming language that is class-based, object-oriented, and designed to have as few implementation dependencies as possible. It follows the *Write once, run anywhere* slogan which illustrates its cross-platform benefits. Ideally, this means that a Java program could be developed on any device, compiled into standard bytecode, and be expected to run on any device equipped with a Java virtual machine (JVM). The syntax of Java is similar to C and C++. The latest versions are Java 14, released in March 2020, and Java 11, a currently supported long-term support (LTS) version, released on September 25, 2018.

These are the five primary goals for creating Java:

- Simple, object-oriented, and familiar.
- Robust and secure.
- Portable.
- High performance.
- Interpreted, threaded, and dynamic.

2.5.1. Java Virtual Machine

One design goal of Java is portability, which means that programs written in Java must run on any other device supporting the JVM. This is achieved by compiling the Java language code to an intermediate called *Java bytecode*, instead of directly to the specific machine code. Java bytecode instructions are analogous to machine code, but they are intended to be executed on a virtual machine (VM) written specifically for the host hardware. End users commonly use a Java Runtime Environment (JRE) installed on their machine for standalone Java applications. The use of universal bytecode makes the program portable. However, the overhead of interpreting bytecode into machine instructions made interpreted programs almost always run more slowly than native executables.

2.5.2. Syntax

The syntax of Java is largely influenced by C++. Unlike C++, which combines the syntax for structured, generic, and object-oriented programming, Java was built as an object-oriented language. All code is written inside classes, and every data item is an object, with the exception of the primitive data types, (i.e. integers, floating-point numbers, boolean values, and characters), which are not objects for performance reasons. Java uses comments similar to those of C++.

2.5.3. Java Servlets

A servlet is a Java Programming language class that is used to create a dynamic website; a dynamic website has the capability to change the site contents according to the time or are able to generate the contents according to the request received by the client.

Servlets are Java programs that run on the Java web server. They are used to handle the request obtained from the web server, process the request, produce the response, then send response back to the web server. Figure 2.13 helps understand this concept.

1. The clients send the request to the web server.
2. The web server receives the request.
3. The web server passes the request to the corresponding servlet.
4. The servlet processes the request and generates the response in the form of output.
5. The servlet sends the response back to the web server.
6. The web server sends the response back to the client and the client browser displays it on the screen.

The server-side extensions are the technologies that are used to create dynamic Web pages. To provide the facility of dynamic Web pages they need a container or Web server. APIs are used for this. These APIs allow us to build programs that can run with a Web server. In this case, Java Servlet is also one of the component APIs of Java Platform Enterprise Edition which sets standards for creating dynamic Web applications in Java [24].

1. It reads the explicit data sent by the browser. This can be an HTML web page. It also reads the implicit HTTP request data sent by the browser. This can include cookies, media types and compression schemes the browser understands, and so forth.

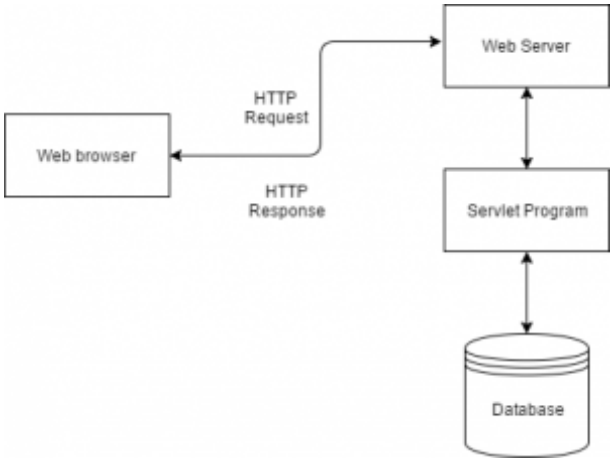

Figure 2.13: Servlet Architecture (Image found in [24])

2. After that, the servlet processes the data and generates the results. This process may require communicating with a database, invoking a Web service, or computing the response directly.

3. After processing, it sends the explicit data to the browser. This document can be sent in a variety of formats, mostly in HTML or XML.

4. Finally, it also sends the implicit HTTP response to the browser. This includes telling the client what type of document is being returned.

2.5.3.1. Create a Servlet

The steps to create a servlet are the following [25]:

1. Create a directory structure.

2. Create a Servlet.

3. Compile the Servlet.

4. Add mappings to the *web.xml* file.

5. Start the server and deploy the project.

6. Access the Servlet.

There are three files needed for any servlet program *index.html* file (Listing 2.1), Java class file (Listing 2.2), and *web.xml* file (Listing 2.3).

```
<!DOCTYPE HTML>
<html>
  <body>
    <form action = "add">
      Enter 1st number: <input type = "text" name = "num1">
      Enter 2nd number: <input type = "text" name = "num2">
    </form>
  </body>
</html>
```

Listing 2.1: index.html (Code found in [25])

```java
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Add extends HttpServlet{

    public void service(HttpServletRequest req, HttpServletResponse res)
        throws IOException
    {
        int i = Integer.parseInt(req.getParameter("num1"));
        int j = Integer.parseInt(req.getParameter("num2"));
        int k = i + j;
        PrintWriter out = res.getWriter();
        out.println("Result is " + k);
    }
}
```

Listing 2.2: Java Servlet Example (Code found in [25])

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<web-app xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance" version =
    "3.0">
  <display-name>Basic</display-name>
  <servlet>
    <servlet-name>Addition</servlet-name>
    <servlet-class>edureka.Add</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Addition</servlet-name>
    <url-pattern>/add</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Listing 2.3: web.xml (Code found in [25])

2.6. JavaServer Pages [26]

JSP is a technology that allows developers to add dynamic content to web pages. Without JSP, updating the appearance or the content of plain static HTML pages would have to be done by hand. With JSP, content on a website can be dependent on many factors, including the time of the day, the information provided by the user, the user's history of interaction with your web site, and even the user's browser type. This capability is essential to provide online services in which it is possible to tailor each response to the viewer who made the request, depending on the viewer's preferences and requirements. A crucial aspect of providing meaningful online services is to be able to remember data associated with the service and its users. That is why databases play an essential role in dynamic web pages.

With JSP, the web page does not exist on the server, the server creates it when it responds to each request.

A java Servlet queries a database containing the information needed and formats an HTML page with that data. Plain HTML cannot make queries on a database, but Java can, so JSP has the means to include snippets of Java inside an HTML page.

These steps explain how the web server creates the web page with the help of figure 2.14:

1. Like with a normal page, the browser sends an HTTP request to the web server. This does not change with JSP, although the URL probably ends in .jsp instead of .html.
2. The web server is a Java server. The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine.
3. The JSP engine loads the JSP page from disk and converts it into a Java Servlet.
4. The JSP engine compiles the servlet into an executable class and forwards the original request to another part of the web server called the servlet engine.
5. The servlet engine loads the servlet class and executes it. During execution, the servlet produces an output in HTML format, which the servlet engine passes to the web server inside an HTTP response.
6. The web server forwards the HTTP response to your browser.
7. The web browser handles the dynamically generated HTML page inside the HTTP response exactly as if it were a static page.

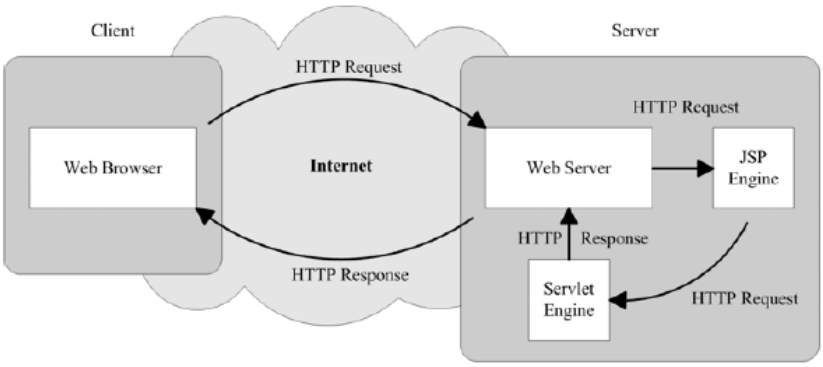

Figure 2.14: How the server creates the page with JSP (Image found in [27])

```
1 <% @page language = "java" contentType = "text/html" %>
2 <html>
3   <head>
4     <title>Hello World dynamic HTML</title>
5   </head>
6   <body>
7     Hello World!
8     <% out.println("<br/>Your IP address is " + request.getRemoteAddr());
9       String userAgent = request.getHeader("user-agent");
10      String browser = "unknown";
11      out.print("<br/>and your browser is ");
12      if (-1 < userAgent.indexOf("MSIE")) {
13        browser = "MS Internet Explorer";
14      }
15      else if (-1 < userAgent.indexOf("Firefox")) {
```

```
16          browser = "Mozilla Firefox";
17        }
18        else if (-1 < userAgent.indexOf("Opera")) {
19          browser = "Opera";
20        }
21        else if (-1 < userAgent.indexOf("Chrome")) {
22          browser = "Google Chrome";
23        }
24        else if (-1 < userAgent.indexOf("Safari")) {
25          browser = "Apple Safari";
26        }
27      }
28      out.println(browser); %>
29    </body>
30  </html>
```

Listing 2.4: Java Code from the Hello World! JSP Page (Code found in [27])

The code in listing 2.4 renders the view in figure 2.15:

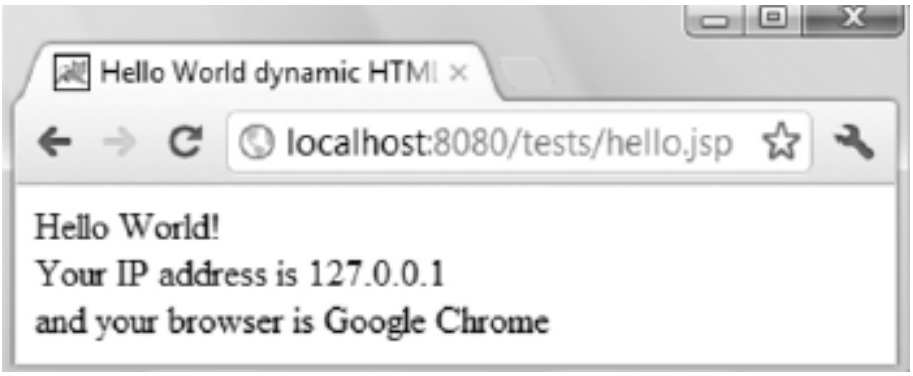

Figure 2.15: Hello World in JSP with Google Chrome (Image found in [27])

2.6.1. JSP Elements [27]

A JSP page is made out of a page template, which consists of HTML code and JSP elements such as scripting elements, directive elements, and action elements.

Scripting elements (scriptlets) consist of code delimited by particular sequences of characters. They are delimited by the pair <% and %>. All scripting elements are Java fragments capable of manipulating Java objects, invoking their methods and catching Java exceptions. They can send data to the output, and they execute when the page is requested.

Directive elements are messages to the server. They provide information on the page itself necessary for its translation. They have no association with each individual request, so they do not output any text to the HTML response. Other directives available are include and taglib. This data influences the translation process from a script file to a Java servlet class. As directives only play a role when a JSP page is re-compiled after its been modified, they have no specific effect on the individual HTML responses.

There are three directives that can be used in JSP pages: *page*, *include*, and *taglib*. Their syntax is as follows: `<% @directive-name attr1="v1" [attr2="v2"...] %>`

The page directive defines several page-dependent properties expressed through attributes. These properties should appear only once in a JSP page. This directive is used in all JSP

pages. Typically, a JSP page starts with a page directive to tell the server that the scripting language is Java and that the output is to be HTML: `<% @page language="java" contentType="text/html" %>`

2.6.1.1. The include Directive

The include directive allows to insert into a JSP page the unprocessed content of another text file. `<% @include file"some_jsp_code.jspf" %>`[1] The server does the merging before any translation, so the content of the included file is pasted into the page. All the HTML tags and JSP variables defined before the line containing the directive are available to the included code.

2.6.1.2. The taglib Directive

The JSP tags available for use can be extended by directing the server to use external self-contained tag libraries. The taglib directory identifies a tag library and specifies what prefix to use to identify its tags.

The code `<% @taglib uri="http://mysite.com/mytags" prefix="my" %>` makes it possible to write the following line as part of the JSP page:

`<my:oneOfMyTags>...</my:oneOfMyTags>`.

The following code includes the core JSP Standard Tag Library:

`<% @taglib uri="http://java.sun.com/jsp/jstl/core" prefix="x" %>`.

Scripting elements allow Java code to be embeded in an HTML page. Java methods consist of a sequence of operations to instantiate objects, allocate memory for variables, calculate expressions, perform assignments, etc.

2.6.1.3. Expressions

An expression scripting element inserts into the page the result of a Java expression enclosed in the pair `<%=` and `%>`. For example, in the code in listing 2.5, the expression scripting element inserts the current date into the generated HTML page:

```
1 <% @page import = "java.util.Date" %>
2 Server date and time: <%= new Date() %>
```

Listing 2.5: expression scriplet example (Code found in [27])

It is possible to use any Java expression within a Scriplet as long as it provides a value. This means that every Java expression will do, except a void method. For example, `<%= (condition) ? "yes" : "no" %>` is valid, because it is evaluated to a string.

The out object is used in JSP like the `System.out` object in Java: to write to the standard output. The standard output for a JSP page is the body of the HTML response sent back to the client. Therefore, the scriptlet `<% out.print(expression) %>` causes the result of the expression to be displayed in the client's browser. The same result can be achieved by typing `<%= expression %>`. Listing 2.6 is a good example.

```
1 <% out.print(user.toString()) %>
2 <%= user.toString() %>
```

Listing 2.6: The out object (Code found in [27])

[1] JSPF stands for JSP Fragment

2.6.2. JSP Standard Actions

Action elements specify activities that, like the scripting elements, need to be performed when the page is requested, because their purpose is to encapsulate activities that the server performs when handling an HTTP request from a client. Action elements can use, modify, and/or create objects, and they may affect the way data is shown to the user. They normally take the following form: `<jsp:action-name attr1="v1" [attr2="v2"...]>...</jsp:action-name>` [28].

There are eight JSP standard actions (`forward`, `include`, `useBean`, `setProperty`, `getProperty`, `text`, `element`, and `plugin`) and five additional actions that can only appear in the body of other actions (param, params, attribute, body, and fallback).

2.6.2.1. Forward

The forward action makes possible to abort execution of the current page and transfer the request to another page, listings 2.7 and 2.8 show how to create the view in figure 2.16, note the title in the tab [29]:

```
<html>
  <head>
    <title>JSP forward example with parameters</title>
  </head>
  <body>
    <jsp:forward page = "display.jsp">
    <jsp:param name = "name" value = "Chaitanya" />
    <jsp:param name = "site" value = "BeginnersBook.com" />
    <jsp:param name = "tutorialname" value = "jsp forward action" />
    <jsp:param name = "reqcamefrom" value = "index.jsp" />
    </jsp:forward>
  </body>
</html>
```

Listing 2.7: forward Scriplet example (Code found in [29])

```
<html>
  <head>
    <title>Display Page</title>
  </head>
  <body>
    <h2>Hello this is a display.jsp Page</h2>
    My name is: <%= request.getParameter("name") %><br>
    Website: <%= request.getParameter("site") %><br>
    Topic: <%= request.getParameter("tutorialname") %><br>
    Forward Request came from the page:
    <%= request.getParameter("reqcamefrom") %>
  </body>
</html>
```

Listing 2.8: forward Scriplet result (Code found in [29])

2.6.2.2. Include

When the include action is executed, the content is not replaced, instead it *includes* the new content into the old one [30]. Listings 2.9 and 2.10 have the code necessary to generate the view in figure 2.17.

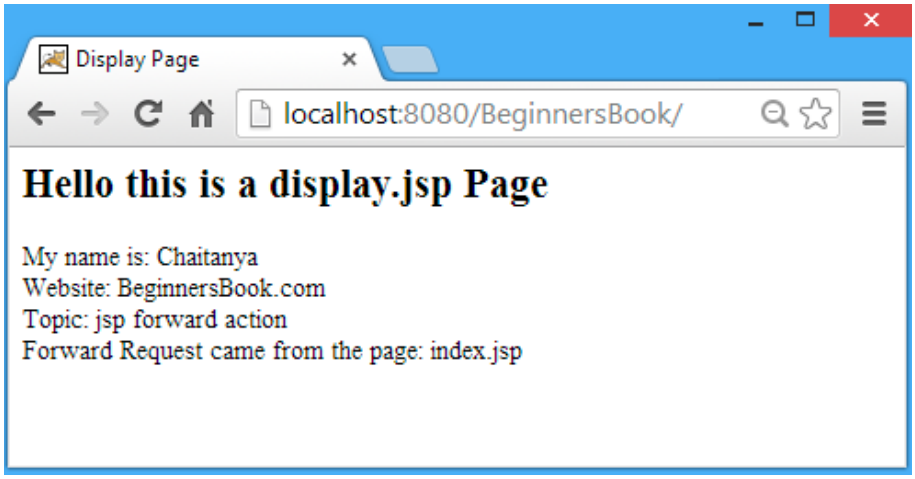

Figure 2.16: Forward display example code (Image found in [29])

```
<html>
  <head>
    <title>JSP Include example with parameters</title>
  </head>
  <body>
    <h2>This is index.jsp Page</h2>
    <jsp:include page = "display.jsp" >
    <jsp:param name = "userid" value = "Chaitanya" />
    <jsp:param name = "password" value = "Chaitanya" />
    <jsp:param name = "name" value = "Chaitanya Pratap Singh" />
    <jsp:param name = "age" value = "27" />
    </jsp:include>
  </body>
</html>
```

Listing 2.9: index.jsp (Code found in [30])

```
<html>
  <head>
    <title>Display Page</title>
  </head>
  <body>
    <h2>Hello this is a display.jsp Page</h2>
    UserID: <%= request.getParameter("userid") %><br>
    Password is: <%= request.getParameter("password") %><br>
    User Name: <%= request.getParameter("name") %><br>
    Age: <%= request.getParameter("age") %>
  </body>
</html>
```

Listing 2.10: display.jsp (Code found in [30])

2.6.2.3. useBean, setProperty, getProperty

A good example on how to use these properties is this [31]:

Listing 2.11 is a bean class *Details* where there are three variables: username, age and password. In order to use the bean class and its properties in JSP the class must be initialized.

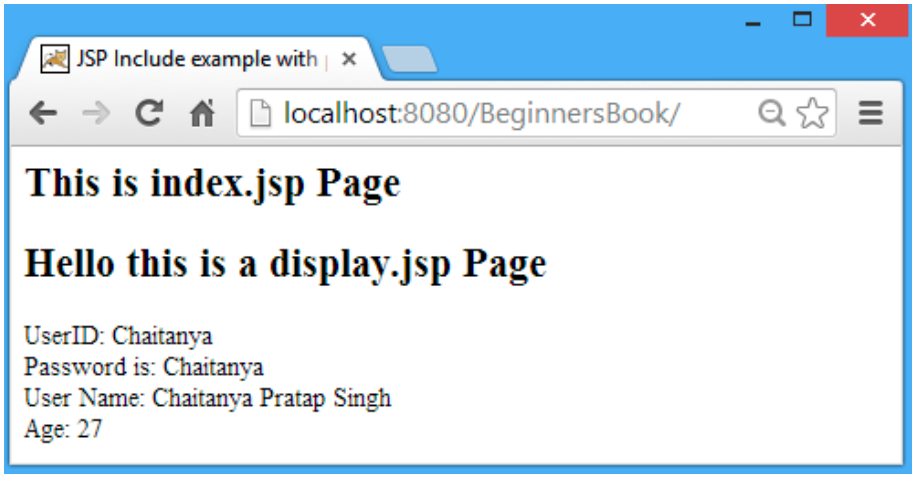

Figure 2.17: Include display example code (Image found in [30])

```
1  public class Details {
2
3      private int age;
4      private String username;
5      private String password;
6
7      public Details() {
8      }
9
10     public String getUsername() {
11         return username;
12     }
13
14     public void setUsername(String username) {
15         this.username = username;
16     }
17
18     public int getAge() {
19         return age;
20     }
21
22     public void setAge(int age) {
23         this.age = age;
24     }
25
26     public String getPassword() {
27         return password;
28     }
29
30     public void setPassword(String password) {
31         this.password = password;
32     }
33 }
```

Listing 2.11: Bean Class (Code found in [31])

The *useBean* action is used to initialize the class. The *setProperty* action is used to set the "*" property to map the values based on their names because it uses the same property name in bean class and index.jsp JSP page. To get the property values, *getProperty* action tag is used. Listings 2.12 and 2.13 are a good example.

```
<jsp:useBean id = "userinfo" class = "beginnersbook.com.Details"></jsp:
    useBean>
<jsp:setProperty property = "*" name = "userinfo"/>
You have enterted below details:<br>
<jsp:getProperty property = "username" name = "userinfo"/><br>
<jsp:getProperty property = "password" name = "userinfo"/><br>
<jsp:getProperty property = "age" name = "userinfo"/>
```

Listing 2.12: useBean (Code found in [31])

```
<html>
  <head>
    <title>useBean, getProperty and setProperty example</title>
  </head>
  <form action = "userdetails.jsp" method = "post">
    User Name: <input type = "text" name = "username"><br>
    User Password: <input type = "password" name = "password"><br>
    User Age: <input type = "text" name = "age"><br>
    <input type = "submit" value = "register">
  </form>
</html>
```

Listing 2.13: index.jsp (Code found in [31])

The result can be seen in figure 2.18.

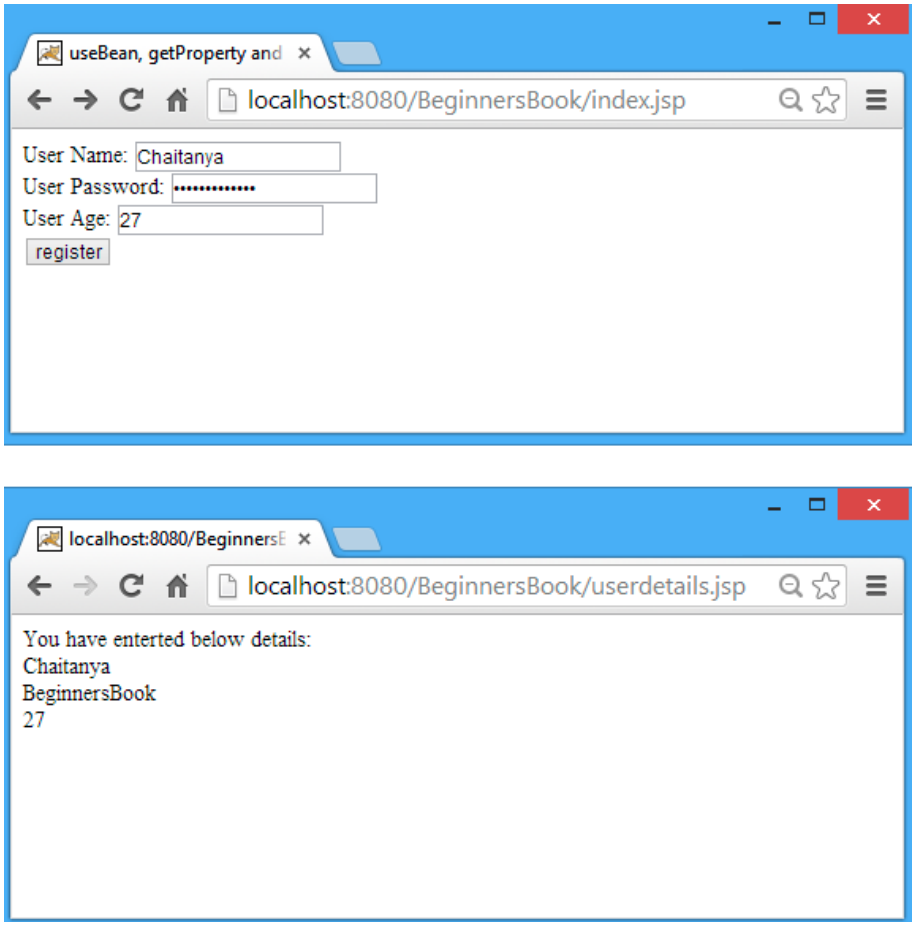

Figure 2.18: useBean, getProperty and setProperty result (Image found in [31])

2.6.2.4. Text

The text action can be used to write template text. Its body cannot contain other elements.

2.6.2.5. Plugin, Params, and Fallback

These three actions are for embeding an object in a web page. In this example, the plugin action (listing 2.14) generates the appropriate browser-dependent HTML construct to embed the applet (listing 2.15).

```
<% @page language = "java" contentType = "text/html" %>
<html>
  <head>
    <title>Action: plugin</title>
  </head>
  <body>
    <jsp:plugin type = "applet" code = "MyApplet.class"
    codebase = "/tests" height = "100" width = "100">
      <jsp:params>
        <jsp:param name = "line" value = "Well said!"/>
      </jsp:params>
      <jsp:fallback>Unable to start plugin</jsp:fallback>
    </jsp:plugin>
  </body>
</html>
```

Listing 2.14: plugin.jsp (Code found in [28])

```
import java.awt.*;
import java.applet.*;

public class MyApplet extends Applet {
    String line;

    public void init() {
        line = getParameter("line");
    }

    public void paint(Graphics page) {
        page.setColor(Color.red);
        page.fillRect(0, 0, 50, 50);
        page.setColor(Color.green);
        page.fillRect(50, 0, 50, 50);
        page.setColor(Color.blue);
        page.fillRect(0, 50, 50, 50);
        page.setColor(Color.yellow);
        page.fillRect(50, 50, 50, 50);
        page.setColor(Color.black);
        page.drawString(line, 10, 40);
    }
}
```

Listing 2.15: MyApplet.java (Code found in [28])

The result can be seen in figure 2.19.

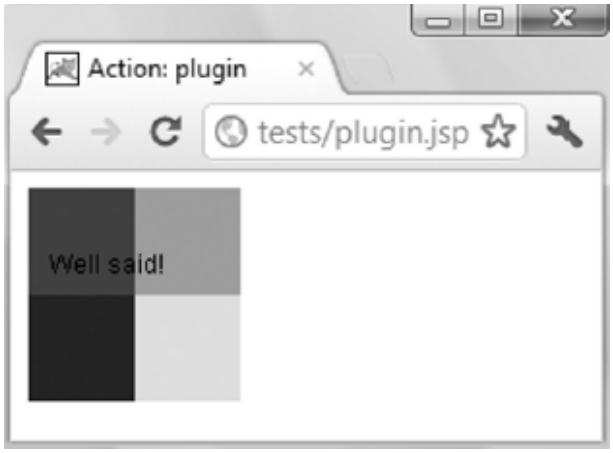

Figure 2.19: JSP plugin action (Image found in [28])

Core	i18n	Functions	Database	XML
c:catch	fmt:bundle	fn:contains	sql:dateParam	x:choose
c:choose	fmt:formatDate	fn:trim	sql:param	x:forEach
c:forEach	fmt:formatNumber	fn:endsWith	sql:query	x:if
c:forTokens	fmt:message	fn:escapeXml	sql:setDataSource	x:otherwise
c:if	fmt:param	fn:indexOf	sql:transaction	x:out
c:import	fmt:parseDate	fn:join	sql:update	x:param
c:otherwise	fmt:parseNumber	fn:length	-	x:parse
c:out	fmt:timeZone	fn:replace	-	x:set
c:param	fmt:setBundle	fn:split	-	x:transform
c:redirect	fmt:setLocale	fn:startsWith	-	x:when
c:remove	fmt:setTimeZone	fn:substring	-	-
c:set	-	fn:substringAfter	-	-
c:url	-	fn:substringBefore	-	-
c:when	-	fn:toLowerCase	-	-
-	-	fn:toUpperCase	-	-

Table 2.1: JSTL Tags (Table found in [32])

2.6.3. JSP Standard Tag Library

JSP also provides a mechanism to define custom actions, in which a custom prefix replaces the prefix jsp of the standard actions. The tag extension mechanism makes possible to create libraries of custom actions, which then can be used in all your applications. This is JSTL, the JSP Standard Tag Library. JSTL consists of five tag libraries. Table 2.1 shows the tags included in each library.

2.7. JavaServer Faces

JavaServer Faces (JSF) was conceived in 2001 and released in 2004 as an attempt to bring a standardized MVC (Model-View-Controller) web framework base for Java.

JSP technology is based on processing a template from start to end, immediately writing to the response as tags are encountered. JSF, on the other hand, requires a phased approach where components need to be able to inspect and act on the component tree, which is built

from the tags on the page, before starting to write anything to the response.

In 2009 JSF 2.0 arrived and JSP was deprecated [33].

JavaServer Pages (JSP) is used to define pages, which are compiled to servlets. JavaServer Faces (JSF) is a complete web MVC framework. It is implemented as a servlet itself.

JSF has these advantages for web development:

- Makes it easy to construct a UI from a set of reusable UI components.
- Simplifies migration of application data to and from the UI.
- Helps manage UI state across server requests.
- Provides a simple model for wiring client-generated events to server-side application code.
- Allows custom UI components to be easily built and reused.

JSF is intended to handle component status across multiple requests. It can be used to process complex forms, even if they span multiple pages. It provides a strongly typed event model for events created on the client side and implements powerful page navigation.

2.7.1. JSF Lifecycle Overview

The JSF lifecycle consists of six phases, ilustrated in figure 2.20:

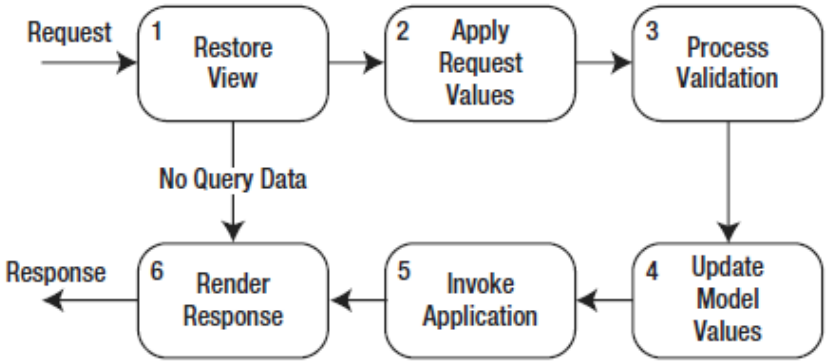

Figure 2.20: JSF Lifecycle (Image found in [32])

1. Restore View: The JSF servlet builds the view of the requested page as a component tree that contains the information associated with all components of the page. If the page is requested for the first time, JSF creates an empty view, wires event handlers and validators to its components, and then saves it in a FacesContext object, before jumping directly to Render Response. By saving the view, JSF makes it possible to repopulate the phase 6 if necessary—for example, when an error ocurrs.
2. Apply Request Values: JSF goes through the component tree and executes each component's decode method, which extracts values from the request parameters, or possibly from cookies or headers. It also automatically converts the parameters that are associated with object properties of nonstring types.

3. Process Validation: The servlet invokes the validate methods for all components of the validators that had been registered during the Restore View. The validation rules can be custom-defined or predefined by JSF. For each validate method that returns false, the servlet marks the component invalid and queues an error message to the FacesContext. At the end of this phase, if there are validation errors, JSF jumps directly to Render Response, so that error messages can be displayed to the user.

4. Update Model Values: During this phase, the values of the components are copied to the corresponding properties of the managed beans that are wired to them. JSF does it by executing the component method *updateModel*, which also performs type conversions when necessary.

5. Invoke Application: During this phase, the servlet processes the application level events by executing the corresponding handlers. When the user submits a form or clicks on a link of a JSF application, the JSF servlet generates a corresponding application level event. One task developers have to do when working on a JSF application is to assign a handler to each one of the possible application events. This is where they specify what should happen next by returning outcomes linked to possible next pages.

6. Render Response: The servlet creates a response component tree and delegates the rendering of the page to the server. Each component renders itself as the server goes through the JSF tags. At the end of this phase, the state of the response is saved so that the servlet can access it during the Restore View phase of subsequent requests to the same page.

2.7.2. JSF Tags

Many HTML tags in JSF are rewritten as table 2.2 shows. In JSP whe had tags. Now in JSF we have Facelets. These are the main tags in JSF:

- `<c:forEach>`, see figure 2.16 and figure 2.17
- `<c:if>`, see figure 2.18
- `<c:set>`, see figure 2.18
- `<c:choose><c:when><c:otherwise>`, see figure 2.19
- `<c:catch>`, see figure 2.20

The code in listing 2.16 creates three separate `<h:outputText>` components in the component tree, like this:

```
<c:forEach items = "#{bean.items}" var = "item">
  <h:outputText id = "item_#{item.id}" value = "#{item.value}" />
</c:forEach>
```

Listing 2.16: `<c:forEach>` example (Code found in [32])

```
<h:outputText id = "item_1" value = "#{bean.items[0].value}" />
<h:outputText id = "item_2" value = "#{bean.items[1].value}" />
<h:outputText id = "item_3" value = "#{bean.items[2].value}" />
```

Listing 2.17: `<c:forEach>` result (Code found in [32])

Tag Name	HTML Element
h:body	body
h:button	input type="button"
h:column	–
h:commandButton	input type="submit"
h:commandLink	a
h:dataTable	table
h:doctype	<!DOCTYPE>declaration
h:form	form
h:graphicImage	img
h:head	head
h:inputHidden	input type="hidden"
h:inputSecret	input type="password"
h:inputText	input type="text"
h:inputTextarea	input type="textarea"
h:link	a
h:message	span or text
h:messages	span or text
h:outputFormat	span or text
h:outputLabel	label
h:outputLink	a
h:outputScript	script
h:outputStylesheet	link
h:outputText	span or text
h:panelGrid	table
h:panelGroup	div or span
h:selectBooleanCheckbox	input type="checkbox"
h:selectManyCheckbox	multiple input type="checkbox"
h:selectManyListbox	select and multiple option
h:selectManyMenu	select and multiple option
h:selectOneListbox	select and multiple option
h:selectOneMenu	select and multiple option
h:selectOneRadio	multiple input type="radio"

Table 2.2: HTML Tags and HTML Elements (Table found in [32])

```
<c:set var = "salary" scope = "session" value = "${2000 * 2}"/>
<c:if test = "${salary > 2000}">
  <p>My salary is: <c:out value = "${salary}"/><p>
</c:if>
```

Listing 2.18: <c:set> and <c:if> example (Code found in [32])

```
<c:choose>
  <c:when test = "#{type eq 'password'}">
    <h:inputSecret id = "#{id}" label = "#{label}" value = "#{value}" />
  </c:when>
  <c:when test = "#{type eq 'textarea'}">
    <h:inputTextarea id = "#{id}" label = "#{label}" value = "#{value}" />
  </c:when>
  <c:otherwise>
    <h:inputText id = "#{id}" label = "#{label}" value = "#{value}"
      a:type = "#{type}">
    </h:inputText>
  </c:otherwise>
</c:choose>
```

Listing 2.19: <c:choose> example (Code found in [32])

```
<c:catch var = "catchException">
  <% int x = 5 / 0 %>
</c:catch>

<c:if test = "${catchException != null}">
  <p>The exception is : ${catchException} <br />
    There is an exception: ${catchException.message}
  </p>
</c:if>
```

Listing 2.20: <c:catch> example (Code found in [32])

2.7.3. Component Tree

When JSF processes a request, it searches for the requested page and scans its content. Besides HTML, the page can contain special tags, like <h:inputText value="..." />. Those elements have a special XML namespace that indicates they must be handled by JSF. These UI elements are collected into a tree data structure. Figure 2.21 is the component tree that is restored in the restore view phase, it shows how the tree of listing 2.21 looks and listings 2.22 and 2.23 show how to manipulate it.

The root of this tree always is UIViewRoot. Below that, there are two siblings representing the head and body of that page, and below that, the other elements nested within that page. In this example, all elements with the special tags of the JSF/HTML namespace (with the prefix h:) show up in the Component Tree. If this namespace is omitted, these elements will be treated as pure HTML and won't be included in the component tree.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>
<html xmlns = "http://www.w3.org/1999/xhtml"
  xmlns:h = "http://xmlns.jcp.org/jsf/html">
  <h:head>
    <h:outputLabel value = "Demo"/>
  </h:head>
  <h:body>
```

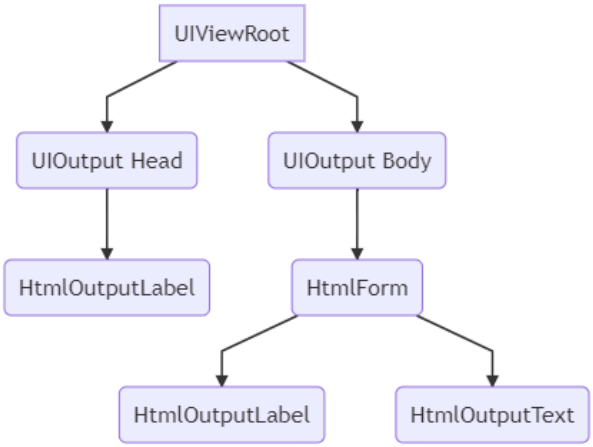

Figure 2.21: Component Tree

```
9       <h:form>
10        <h:outputLabel value = "Param1: "/>
11        <h:inputText value = "#{tinyCalculator.param1}"/>
12      </h:form>
13    </h:body>
14  </html>
```

Listing 2.21: Component Tree example (Code found in [34])

2.7.3.1. Manipulating the Component Tree

Manipulating the Component Tree can be done declaratively using JSTL tags as well as programmatically using Java code. Tree-based hierarchies in code are best readable and maintainable when using a hierarchical markup language such as XML. Facelets is already XML based. JSTL is also XML based and therefore can be integrated in a Facelets file. JSTL is therefore the recommended approach to dynamically manipulate the component tree, rather than Java code.

In other words, the earliest moment when you can guarantee safely modifying the component tree is during the *PostAddToViewEvent* and the latest moment when you can guarantee safely modifying the component tree is during the *PreRenderViewEvent*. Any moment in between is thus also possible. So manipulating the component tree during the render response phase (sixth phase) is a bad idea.

```
1  <h:form id = "dynamicFormId">
2    <f:event type = "postAddToView" listener = "#{dynamicForm.populate}" />
3  </h:form>
```

Listing 2.22: Example on how to manipulate the Component Tree (Code found in [34])

Where the `dynamicForm` looks something like listing 2.23:

```
1  Named RequestScoped
2  public class DynamicForm {
3
4      private transient UIForm form;
5
6      private Map<String, Object> values = new HashMap<>();
7
8      private FieldService fieldService;
```

```
9
10      public void populate(ComponentSystemEvent event) {
11          form = (UIForm) event.getComponent();
12          List<Field> fields = fieldService.list(form.getId());
13          fields.forEach(field -> field.populate(this));
14      }
15
16      public void createOutputLabel(Field field) {
17          HtmlOutputLabel label = new HtmlOutputLabel();
18          label.setId(field.getName() + "_l");
19          label.setFor(field.getName());
20          label.setValue(field.getLabel());
21          form.getChildren().add(label);
22      }
23
24      public void createInputText(Field field) {
25          HtmlInputText text = new HtmlInputText();
26          text.setId(field.getName());
27          text.setLabel(field.getLabel());
28          text.setValueExpression("value", createValueExpression(field));
29          form.getChildren().add(text);
30      }
31
32      public void createMessage(Field field) {
33          HtmlMessage message = new HtmlMessage();
34          message.setId(field.getName() + "_m");
35          message.setFor(field.getName());
36          form.getChildren().add(message);
37      }
38
39      public static ValueExpression createValueExpression(Field field) {
40          String el = "#{dynamicForm.values['" + field.getName() + "']}"
41          FacesContext context = FacesContext.getCurrentInstance();
42          ELContext elContext = context.getELContext();
43          return context.getApplication().getExpressionFactory()
44          .createValueExpression(elContext, el, Object.class);
45      }
46
47      public Map<String, Object> getValues() {
48          return values;
49      }
50  }
```

Listing 2.23: Example on how to manipulate the Component Tree (Code found in [34])

And where the abstract class Field represents the custom model of a form field with at least type, name, and label properties and the implementation of a `TextField#populate()` looks something like listing 2.24:

```
1  public void populate(DynamicFormBean form) {
2      form.createOutputLabel(this);
3      form.createInputText(this);
4      form.createMessage(this);
5  }
```

Listing 2.24: Example on how to populate the Component Tree (Code found in [34])

2.7.4. Conversion and Validation [34]

As an MVC framework, JSF needs to convert between Java objects and strings all the time. The HTTP request is basically broken down into plain vanilla strings representing headers and parameters, not as Java objects. The HTTP response is written as one big sequence of characters representing HTML or XML, not as a Java object. However, the average Java model behind a JSF page does not necessarily contain only String properties. That is why Converters come into the picture: converting between objects in model and strings in view.

Before updating the model values with freshly submitted and converted values, the program must validate whether they comply with the business rules of the web application and, if necessary, present end users an error message so that they can fix any errors themselves. Usually, the business rules are already very well defined in the data store, such as a relational database management system. The Frontend should make absolutely sure that the submitted and converted values can be inserted in the database without errors.

The standard converters are these:

- `<f:convertNumber>`, see listing 2.25
- `<f:convertDateTime>`, see listing 2.26

```
<f:convertNumber type = "number" />
<f:convertNumber type = "currency" />
<f:convertNumber type = "percent" />

<h:outputText value = "#{product.price}">
  <f:convertNumber type = "currency" locale = "en_US" />
</h:outputText>
```

Listing 2.25: Example of `f:convertNumber` (Code found in [34])

```
<f:convertDateTime type = "localDate" pattern = "yyyy-MM-dd" />
<f:convertDateTime type = "localTime" pattern = "HH:mm:ss" />
<f:convertDateTime type = "localDateTime" pattern = "yyyy-MM-dd HH:mm:ss" /
    >
<f:convertDateTime type = "offsetTime" />
<f:convertDateTime type = "offsetDateTime" />
<f:convertDateTime type = "zonedDateTime" />

<h:inputText id = "date" value = "#{bean.date}">
  <f:convertDateTime type = "date" pattern = "yyyy-MM-dd" />
</h:inputText>
```

Listing 2.26: Example of `f:convertDateTime` (Code found in [34])

When the submitted value is successfully converted, then JSF will perform validations on the converted value. It has standard validators of its own.

- `<f:validateLongRange>`/`<f:validateDoubleRange>`, see listing 2.27
- `<f:validateLength>`/`<f:validateRegex>`, see listing 2.28

```
<h:inputText value = "#{bean.quantity}">
  <f:validateLongRange minimum = "1" maximum = "10" />
</h:inputText>
<h:inputText value = "#{bean.quantity}" a:type = "number" a:min = "1"
```

```
5   a:max = "10">
6   <f:validateLongRange minimum = "1" maximum = "10" />
7 </h:inputText>
```

Listing 2.27: Example of `f:validateLongRange`. These validators specify a minimum and/or maximum number value for an input (Code found in [34])

```
1 <h:inputText value = "#{bean.someStringOrInteger}" maxlength = "3">
2   <f:validateLength minimum = "3" maximum = "3" />
3 </h:inputText>
4 <h:inputText value = "#{bean.someString}" maxlength = "3">
5   <f:validateRegex pattern = "[0-9]{3}" />
6 </h:inputText>
```

Listing 2.28: Example of `f:validateRegex` for a fixed length of 3. It converts the value to string and then validate the length of the string. It casts the value to String and then check if the string matches() returns true for the specified pattern attribute (Code found in [34])

2.8. Migrating from Legacy Systems

A legacy system is a system built on what has become obsolete technology. It continues to be used due to the cost of replacing or redesigning it and often despite its poor competitiveness, inability to meet changing business needs and incompatibility with modern equivalents. The implication is that the system is large, monolithic, and difficult to modify, integrate and support [35].

For many businesses, legacy system migrations are becoming an increasingly important issue when they examine their technology portfolios. Legacy apps have become one of the enduring points of pain, as the cost and difficulty in managing them leads to companies needing to rationalize them on regular basis.

The whole prospect of how to exploit the cloud has added pressure on how to migrate a legacy system. The opportunities the cloud presents for increasing automation, enabling rapid innovation, and reducing operational costs are perks in migrating the system.

2.8.1. Problems with Migrating Legacy Applications

Legacy system migration projects bring up the question of which applications to kill off and which to move.

From a business perspective, the challenge is to decide whether a better app will really help the business. From a technical perspective, the question often becomes how to migrate a system with a monolithic architecture to a more modern microservice structure.

To make it possible to evolve monolithic applications, innovate, and allow their capabilities to be used by other applications, the system must be broken apart into simpler component parts [36].

2.8.2. Steps to Follow for Migrating Legacy Systems [37]

- Document the Existing System

  In the Software Development Life Cycle, a key part of the process is gathering requirements from users on the functionality that they need in the application that will

be built. In the case of migrating a legacy system, an analysis on the current system is needed to understand how it works, who uses it, what they use it for.

A good starting point for gathering this information is in the existing documentation for the system that is to be migrated. Often this information is missing or incomplete with legacy systems.

There can be crucial information in guides, manuals, tutorials, training materials, etc. that end users may have used. Most often this type of material provides background information on the functionality but may not provide details of how the underlying processes work.

- Requirements Analysis

  The next step in the migration process is to perform a detailed requirements analysis to determine what features are required in the new application. The features and functionality that are in the legacy system should be included, and also interviews with the end users to know what new features are expected/needed.

  This area of the migration is frequently where the cost and scope of the project can increase if mismanaged. The move from a legacy system to another platform is driven by the request for new features that a legacy application cannot provide or where the cost of providing these features would be very high.

  The most effective way of moving to a new platform is to ensure that the application supports the existing features and functionality first before adding anything new. It may be tempting to add features thinking "while we are at it" but this can be a point of failure if those new features slow or halt development of core functionality that the business requires.

- Platform Selection

  The major decision to be made on the application platform is whether the migrated application will be a client application, installed locally on the user's computer or whether it will be a Web application that the users can run from their browser. Current trends lean toward replacing desktop applications with web-based versions, even though there may be some functionalities that can not be easily replicated in a Web application.

  The programming language must be selected too. There may be a corporate standard development environment or it could be an opportunity to establish a new project. When considering a programming platform, servers cost must be included in the evaluation.

  From the data side, the new database platform must support the new application, as well as hold any existing data migrated over from the legacy system.

- Design, Development and Deployment

  Finally, with the system requirements collected and the application and database platform, then the detailed design and development phase of the migration begins.

  A key to a successful migration is having a thorough detailed design that encompasses the current functionality in the legacy system and user requirements. Users should be able to take the screen designs from your detailed designs and walk through the tasks they would normally carry out in the legacy system that is being replaced. Test migrations of data should be run frequently to ensure that the new application is ready to go live by the release date.

Chapter 3

Solution Design

Summary: In this chapter the proposed solution and the software implementation are described, as well as the development process, the decision making, the used packages and the setbacks that happened along the way.

3.1. Architecture of the Proposed Solution

Chain of Responsibility is a behavioral design pattern that consists on passing a request along the chain of potential handlers until one of them handles the request. This pattern is used in the project as there are many classes that transform some parts of the HTML document into a JSF valid syntax, but all transformers behave similarly, so they inherit from a given class.

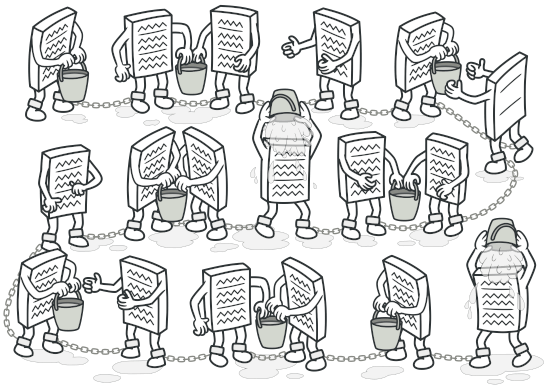

Figure 3.1: Chain of Responsibility (Image found in [38])

The solution developed for the project includes 14 java files, where 13 are components of the Chain of Responsibility and one that has the main method, uses all the transformations and creates a new file with the `.xhtml` extension with the solution for the user to check. It also creates a `.txt` file that contains warning notes on the proposed solution.

The UML diagram for the program developed can be seen in figure 3.2.

These are the files that are included in the developed software. Also there is a JSON file that has all the equivalences between JSF and HTML. There is almost one file for each HTML tag that has a complex transformation:

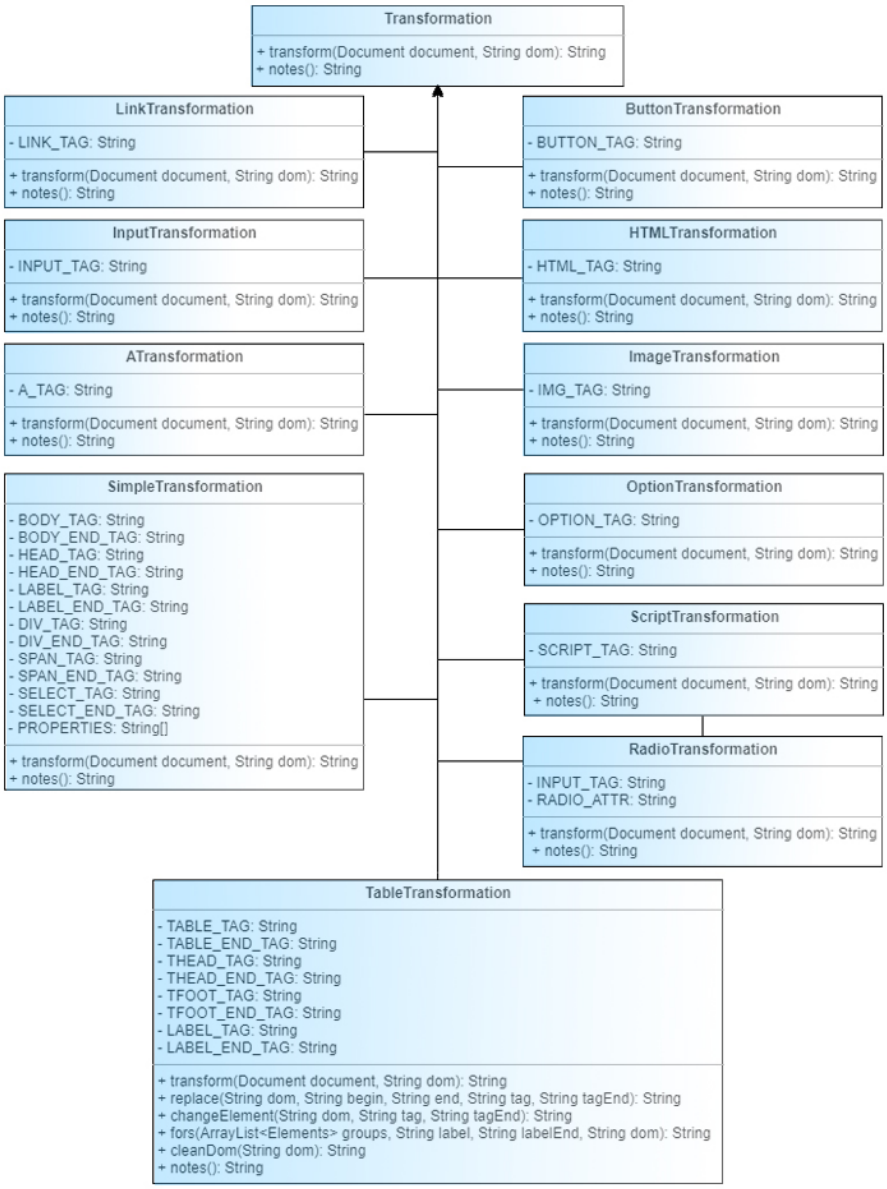

Figure 3.2: UML Diagram for the program

- `Transformation.java`

  This is the file that the rest of the transformations extend to.

- `ATransformation.java`

  This file replaces the tag `a` for `h:outputLink`, the attribute `href` for `value` and moves the text of the tag to the attribute `outcome`, so there is no outside text in this tag.

- `ButtonTransformation.java`

  This file replaces the tag `button` for `h:commandButton`.

- `HTMLTransformation.java`

  This file adds the attributes `xmlns` with the value `http://www.w3.org/1999/xhtml` and `xmlns:h` with the value `http://xmlns.jcp.org/jsf/html` to the `html` tag.

- `ImageTransformation.java`

  This file replaces the tag `img` for `h:graphicImage`, and the attribute `src` for `value`.

- `InputTransformation.java`

  This file replaces the tag `input` with a set of JSF tags depending on the value of the `type` attribute. These are the set of tags contemplated for the transformation:

  - `hidden` for `h:inputHidden`
  - `checkbox` for `h:selectBooleanCheckbox`
  - `password` for `h:inputSecret`
  - `radio` for `h:selectOneRadio`
  - `text` for `h:inputText`
  - `submit` for `h:commandButton`

  Every one of them produces a different JSF tag, but the attributes for those tags remain the same.

- `LinkTransformation.java`

  This file replaces the tag `link` for `h:outputStylesheet`, the attribute `type` for `library`, `href` for `name` and removes the closing tag.

- `OptionTransformation.java`

  This file replaces the tag `option` for `f:selectItem`, the attribute `value` for `value` and moves the text of the tag to the attribute `itemLabel`, so there is no outside text in this tag.

- `RadioTransformation.java`

  This file replaces the tag `radio` for `f:selectItem`, the attribute `value` for `itemValue`, the surrounding `label` tag is removed, the text inside that tag goes to the `itemLabel` attribute and the tag `select` for `h:selectOneRadio`.

- `ScriptTransformation.java`

  This file replaces the tag `script` for `h:outputScript`, the attribute `type` for `library`, `src` for `name` and removes the closing tag.

- `SimpleTransformation.java`

  This file replaces the following tags with their respective ones. They are very similar and that is why they are all on the same transformation:

  - `body` and `/body` for `h:body` and `/h:body`
  - `head` and `/head` for `h:head` and `/h:head`
  - `form` and `/form` for `h:form` and `/h:form`
  - `label` and `/label` for `h:outputLabel` and `/h:outputLabel`
  - `div` and `/div` for `h:panelGroup` and `/h:panelGroup`
  - `span` and `/span` for `h:outputText` and `/h:outputText`
  - `select` and `/select` for `h:span` and `/h:span`

- `TableTransformation.java`

  This file replaces the `table` tag for `h:panelGrid` and the `thead` and `tfoot` tags for `f:facet`. It also removes the `tbody` and `tr` tags and replaces the `td` tags with `h:outputLabel` tags. It also calculates the column span and puts it as an attribute in the `h:panelGrid` tag. All the other attributes remain the same. There are no explicit rows contemplated in JSF tables.

- `JavaTransformation.java`

  This file contemplates the transformations needed for the raw Java code included in the JSP file. Listing 3.1 groups all the code between `<%` `%lt;` and `%gt;` `%>` tags with the `collectTags()` method.

  Two key methods in this file are `chooseTransform()` and `forEachTransform()` because they transform the java instructions into JSF valid tags. These methods are used in Listing 3.2.

```
1  private ArrayList<String> collectTags(ArrayList<String> matches,
2                                        ArrayList<String> tags) {
3
4      for (String match : matches) {
5          match = match.replace("&gt;", ">").replace("&lt;", "<");
6          if (!match.contains("for")) {
7              for (String tag : match.split("; ")) {
8                  if (tag.contains("}")) {
9                      for (String braces : tag.split("}")) {
10                         tags.add(braces.trim().isEmpty() ? "}" : braces);
11                     }
12                 } else {
13                     tags.add(tag);
14                 }
15             }
16         } else {
17             tags.add(match);
18         }
19     }
20     return tags;
```

```
}
```

Listing 3.1: collectTags method

Listing 3.2 replaces all the tags previously talked about with a corresponding JSF transformation. The way it works is this:

1. It receives an `ArrayList<String>` with all the Java syntax tags in the file.
2. It loops through `tags` in order to transform each tag.
3. It works with a `Stack` that stores the closing match of the tag it is working on, so when it the tag is a *}* (closing curly bracket) it calculates how many tags it has to pop from the Stack.
4. When it is done looping through the array, the `Stack` is empty.

```
private String replaceTags(ArrayList<String> tags, Stack<String> ends, String
    dom) {
    for (int i = 0; i < tags.size(); i++) {
        System.out.println(tags.get(i));
        if (tags.get(i).contains("if")) {
            String jsfTag = "\n<c:choose>\n\t";
            addToStack(ends, "</c:choose>");
            jsfTag += chooseTransform(tags.get(i), ends);
            dom = dom.replace(tags.get(i), jsfTag);
        }
        if (tags.get(i).contains("for")) {
            dom = forEachTransform(tags.get(i), ends, dom);
        }
        if (tags.get(i).contains("}")) {
            if (!ends.empty()) {
                if (ends.peek().toString().equals("\n</c:when>\n") && !
                    containsTag(tags.subList(i, tags.size()), "if")) {

                    dom = dom.replaceFirst(" } ", ends.pop().toString() + ends.
                        pop().toString());

                } else if (ends.peek().toString().equals("\n</c:otherwise>\n")
                    || (ends.size() == 2 && i == (tags.size() - 1))) {

                    dom = dom.replaceFirst(" } ", ends.pop().toString() + ends.
                        pop().toString());

                } else {
                    dom = dom.replaceFirst(" } ", ends.pop().toString());
                }
            }
        }
        if (tags.get(i).contains("else {")) {
            String tag = tags.get(i).replace("{", "");
            dom = dom.replaceFirst(tag, "<c:otherwise>");
            addToStack(ends, "\n</c:otherwise>\n");
        }
    }
    return dom;
}
```

Listing 3.2: replaceTags method

- `JSPtoJSF.java`

  Listing 3.3 shows the code used to eficiently transform the code with all the files previously described. It parses the document with `Jsoup`, and then every transformer acts on the result of the previous transformation. It also creates a file with notes about the transformed file.

  All transformers have a boolean flag initially set to `false`, so if there is a transformer that does not need to operate, it keeps its flag in `false`, but if there is a transformer that needs to take action because there are tags in the JSP/HTML file that involve that transformer, then te flag turns `true` and the notes file contemplates notes for that transformation. These flags allow us to monitor which phases have actually made changes in the input file.

```
1  public static String domJsoup(JSONObject json, File fileInput) throws
       IOException {
2      Document document = Jsoup.parse(fileInput, "UTF-8");
3
4      Transformation[] transformers = {
5          new HTMLTransformation(json),
6          new LinkTransformation(json),
7          new ScriptTransformation(json),
8          new RadioTransformation(json),
9          new InputTransformation(json),
10         new OptionTransformation(json),
11         new ATransformation(json),
12         new ImageTransformation(json),
13         new ButtonTransformation(json),
14         new TableTransformation(json),
15         new SimpleTransformation(json),
16         new JavaTransformation(json)
17     };
18     String dom = document.toString();
19
20     FileOutputStream name = new FileOutputStream(NOTES_FOLDER + changeExtension
           (file, TXT_EXTENSION));
21
22     PrintStream out = new PrintStream(name);
23     int i = file.getName().lastIndexOf('.');
24     String filename = file.getName().substring(0, i);
25
26     out.println("============================================================");
27     out.println("Recommendations for the file " + filename);
28     out.println("============================================================");
29     out.println("Check tabulations and linebreakes for better readability");
30
31     for (Transformation transformer : transformers) {
32         dom = transformer.transform(document, dom);
33         if (transformer.getFlag()) {
34             out.println(transformer.notes());
35         }
36     }
37     out.close();
38     return customUpperCase(dom);
39 }
```

Listing 3.3: Transformation Code

Listing 3.4 has the main method where it prints the solution in an `xhtml` file.

```
public static void processFile() throws IOException, ParseException {
    if (cmd.hasOption("f")) {
        File file = new File(cmd.getOptionValue("f"));
        message("START!!!");
        String filename = cmd.getOptionValue("f").substring(cmd.getOptionValue(
            "f").lastIndexOf("\\") + 1, cmd.getOptionValue("f").length());

        System.out.println("\n The file to be transformed is: " + filename + "\
            n");

        JSONParser parser = new JSONParser();
        try {
            Reader dictionary = new FileReader(JSON_FILE);
            JSONObject json = (JSONObject) parser.parse(dictionary);
            String res = domJsoup(json, file);
            File folder = new File(TRANSFORMATIONS_FOLDER);
            folder.mkdirs();
            FileOutputStream name = new FileOutputStream(TRANSFORMATIONS_FOLDER
                 + changeExtension(file, XHTML_EXTENSION));

            PrintStream out = new PrintStream(name);
            out.print(res);
            out.close();
        } catch (FileNotFoundException e) {
            System.out.println("\n The file seems to be wrong \n");
        }
        message("DONE!!! ");
    }
}
```

Listing 3.4: Transformation Main

- `dictionary.json`

  This file includes all the different equivalences between HTML and JSF, listing 3.5 is an extract of this file's content.

```
{
    "html": [{
        "xmlns": "http://www.w3.org/1999/xhtml",
        "xmlns:h": "http://xmlns.jcp.org/jsf/html"
    }],
    "head": "h:head",
    "body": "h:body",
    "button": [{
        "button": "h:commandButton"
    }],
    "img": [{
        "img": "h:graphicImage",
        "src": "value"
    }],
    "input": [{
        "type": [{
            "hidden": "h:inputHidden",
            "checkbox": "h:selectBooleanCheckbox",
            "password": "h:inputSecret",
            "radio": "h:selectOneRadio",
            "text": "h:inputText",
            "submit": "h:commandButton"
        }]
```

```
    }],
    "a": [{
        "a": "h:outputLink",
        "href": "value"
    }],
    "option": [{
        "option": "f:selectItem",
        "value": "itemValue"
    }],
    "radio": [{
        "radio": "f:selectItem",
        "value": "itemValue",
        "label": "itemLabel",
        "select": "h:selectOneRadio",
    }],
    "script": [{
        "script": "h:outputScript",
        "type": "library",
        "src": "name"
    }],
    "label": "h:outputLabel",
    "span": "h:outputText",
    "table": "h:panelGrid",
    "thead": "f:facet",
    "tfoot": "f:facet",
}
```

Listing 3.5: Dictionary JSON file

3.2. Setbacks

There were a few setbacks while developing the program that influenced the way the software was developed.

- `Jsoup` does not allow tags that are not contemplated in the HTML standard, for example `h:outputLabel`. Other tried solution was the `JDOM` package, but it showed troubles with the closing tags, because it is intended to work with `xml`, but HTML does have some tags that do not need a matching closing tag.

  To fix this problem, I decided to have a string that had the HTML DOM, so after editing the tag with `Jsoup`, I replaced the corresponding tag in the DOM string, so that I could use almost all `Jsoup` methods and replace in the DOM string what was needed.

- When replacing the attributes, `Jsoup` does not allow for camel case attributes inside tags, such as `itemLabel`, so I had to replace all those attributes with the camel case ones before writing the solution in the file.

- When replacing the Java code, the `<c:if>` tag gave some trouble because the JSF equivalent does not support an `else` tag, so instead, I decided to use the `<c:choose>` and `<c:when>` tags because the `<c:otherwise>` tag can be omitted.

- JSTL tags are compatible with JSF, so those tags were not transformed into a JSF equivalent, which would have been using regular Java code and that is against MVC best practices.

- Because JSF is an MVC framework, all the business logic is intended to be written either in the Controller or the Model, so it was not aggressively addressed. This point is related to the previous one.

3.3. Used Packages

Maven is a software project management tool. Using a project object model (POM), it can manage a project's build, reporting and documentation from a central piece of information [39]. It was used for the development of the project.

These are the Java Packages used for developing the software:

3.3.1. `java.io` [40]

The `java.io` package contains the classes that handle fundamental input and output operations in Java. The `I/O` classes can be grouped as follows:

- Cases that handle object serialization.
- Manipulate files on the local filesystem.
- For reading input/writing output from/to a stream of data.

The classes from `java.io` used in the project are the following:

- `java.io.File`
- `java.io.Reader`
- `java.io.FileReader`
- `java.io.IOException`
- `java.io.FileNotFoundException`

This package helped with handling the test files and creating the solution files and the note files on every transformation.

3.3.2. `org.json.simple` [41]

The package `org.json.simple` is a Java toolkit used to encode or decode JSON text.

- `org.json.simple.JSONObject`
- `org.json.simple.parser.JSONParser`
- `org.json.simple.parser.ParseException`

This package helped with handling the JSON file that contains the dictionary.

3.3.3. `Jsoup` [42]

`Jsoup` is a Java package for working with real-world HTML. It provides an API for fetching URLs and extracting and manipulating data, using the HTML5 DOM methods and CSS selectors. It implements the WHATWG (Web Hypertext Application Technology Working Group) HTML5 specification, and parses HTML to the same DOM as modern browsers do. It can scrape and parse HTML from a URL, file, or string.

The classes from `org.jsoup` used in the project are the following:

- `org.jsoup.Jsoup`
- `org.jsoup.nodes.Document`

This package helped with handling the HTML tags to make it easier to transform into a JSF valid tag.

3.3.4. `java.util` [43]

The `java.util` package contains the collections framework, legacy collection classes, event model, date and time, internationalization, and miscellaneous utility classes.

The classes from `java.util` used in the project are the following:

- `java.util.List`
- `java.util.Stack`
- `java.util.ArrayList`
- `java.util.regex.Matcher`
- `java.util.regex.Pattern`

This package helped with the logic of tranforming the Java raw tags.

3.3.5. `commons.cli` [44]

The `commons.cli` package provides an API for parsing command line options passed to programs. It can also print help messages explaining the options available for a custom command line tool.

The classes from `commons.cli` used in the project are the following:

- `org.apache.commons.cli.CommandLine`
- `org.apache.commons.cli.CommandLineParser`
- `org.apache.commons.cli.DefaultParser`
- `org.apache.commons.cli.HelpFormatter`
- `org.apache.commons.cli.Options`
- `org.apache.commons.cli.ParseException`

This package helped with the logic of the command line.

Chapter 4

Tests and Results

Summary: In this chapter there is a tutorial, also the tests and results are shown and described.

4.1. How To Use the System

The program must be run with the command line. The user needs to be located at the folder that has the `.jar` file. To run the program the user must use the command `java -jar <JAR_FILE>`

Figure 4.1 shows the output when the user does not send any arguments.

```
C:\Users\Personal\Documents\Java Libraries>java -jar jspjsf.jar
usage: From JSP to JSF
 -f <arg>   file path to be transformed
 -n <arg>   folder path for warning notes
 -t <arg>   folder path for transfromed file
```

Figure 4.1: Usage without arguments

Figure 4.2 shows how the program runs when a file is sent with in the command. The way to do it is this: `java -jar <JAR_FILE>-f file`. If the user does not provide the a folder to store the notes and the transformations, the program has (or creates) default folders for that.

It is possible to tell the program where to put the notes and the transformed files. If the folders do not exist, the program creates them. Figure 4.3 shows how the program runs when a file and a place to store either/both folders are sent with in the command. The way to to do it is this: `java -jar <JAR_FILE>-f file -n <NOTES_FOLDER>`.

Figure 4.4 shows the result of running the program. The results will be found in the *transformedFiles* folder and some warning/recommendation notes will be found in the *notesFiles* folder if the user does not provide custom folders, otherwise the results will be found there.

If there is a mistake, the command line will prompt a message like figure 4.5:

```
C:\Users\Personal\Documents\Java Libraries>java -jar jspjsf.jar -f index.jsp
usage: From JSP to JSF
 -f <arg>   file path to be transformed
 -n <arg>   folder path for warning notes
 -t <arg>   folder path for transfromed file
================================================
||                START!!!                    ||
================================================

The file to be transformed is: index.jsp

File will be found in: transformedFiles/ folder

Notes will be found in: notesFiles/ folder

================================================
||                DONE!!!                     ||
================================================
```

Figure 4.2: Usage specifying file

```
C:\Users\Personal\Documents\Java Libraries>java -jar jspjsf.jar -f index.jsp -n notes
usage: From JSP to JSF
 -f <arg>   file path to be transformed
 -n <arg>   folder path for warning notes
 -t <arg>   folder path for transfromed file
================================================
||                START!!!                    ||
================================================

The file to be transformed is: index.jsp

File will be found in: transformedFiles/ folder

Notes will be found in: notes/ folder

================================================
||                DONE!!!                     ||
================================================
```

Figure 4.3: Usage specifying a folder to put the notes

Name	Date modified	Type	Size
notes	6/8/2020 7:02 PM	File folder	
notesFiles	6/8/2020 6:58 PM	File folder	
transform	6/8/2020 7:02 PM	File folder	
transformedFiles	6/8/2020 6:58 PM	File folder	
index.jsp	5/8/2020 1:13 PM	JSP File	3 KB
jspjsf	6/8/2020 6:56 PM	Executable Jar File	1,115 KB

Figure 4.4: Resulting folder

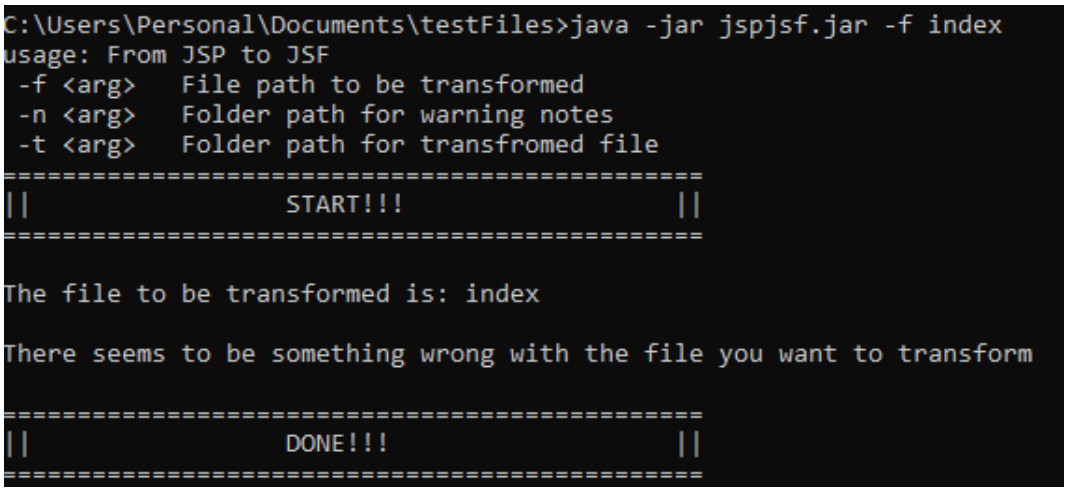

Figure 4.5: Error example

4.2. Results

4.2.1. Select

Listing 4.1 is an example on how a select tag is composed.

```
<select name="example">
  <option value="1">Item 1</option>
  <option value="2">Item 2</option>
</select>
<select name="example">
  <option value="3">Item 3</option>
  <option value="4">Item 4</option>
</select>
```

Listing 4.1: Select example

Listing 4.2 shows how the select tag composition is transformed into `h:selectOneMenu`.

```
<h:selectOneMenu name="example">
  <f:selectItem itemValue="1" itemLabel="Item 1"/>
  <f:selectItem itemValue="2" itemLabel="Item 2"/>
</h:selectOneMenu>
<h:selectOneMenu name="example">
  <f:selectItem itemValue="3" itemLabel="Item 3"/>
  <f:selectItem itemValue="4" itemLabel="Item 4"/>
</h:selectOneMenu>
```

Listing 4.2: `h:selectOneMenu` result

4.2.2. Div and Radio Buttons

Listing 4.3 is an example on how radio buttons are displayed in HTML.

```
<div>
  <input type="radio" id="male" name="gender" value="male">
  <label for="male">Male</label><br>
  <input type="radio" id="female" name="gender" value="female">
  <label for="female">Female</label><br>
  <input type="radio" id="other" name="gender" value="other">
  <label for="other">Other</label>
</div>
```

Listing 4.3: Radio button example

Listing 4.4 shows how radio buttons are transformed into JSF using less tags, making it easier to read.

```
<h:panelGroup>
  <f:selectItem itemLabel="Male" itemValue="male" /><br>
  <f:selectItem itemLabel="Female" itemValue="female" /><br>
  <f:selectItem itemLabel="Other" itemValue="other" />
</h:panelGroup>
```

Listing 4.4: Radio button result

4.2.3. Inputs

Listing 4.5 is an example on different input tags in HTML.

```
<input type="hidden" id="custId" name="custId" value="3487">
<input type="text" id="username" name="username">
<input type="password" id="pass" name="password" minlength="8" required>
```

Listing 4.5: Input tags example

Listing 4.6 shows how those input tags are transformed with more self explanatory tags in JSF.

```
<h:inputHidden id="custId" name="custId" value="3487" />
<h:inputText id="username" name="username" />
<h:inputSecret id="pass" name="password" minlength="8" required />
```

Listing 4.6: Input tags result

4.2.4. Images and Buttons

Listing 4.7 shows how images and buttons are used in HTML.

```
<img src="http://www.tutorialspoint.com/images/jsf-mini-logo.png">
<button type="button">Click Me!</button>
```

Listing 4.7: `img` and `button` tags example

Listing 4.8 shows how images and buttons are transformed in JSF.

```
<h:graphicImage value="http://www.tutorialspoint.com/images/jsf-mini-logo.
    png"/>
<h:commandButton value="Click Me!" />
```

Listing 4.8: Input tags result

4.2.5. Tables

Listing 4.9 shows how tables are created in HTML.

```
  <thead>
    <tr><th colspan = "2" scope = "colgroup">Login</th></tr>
  </thead>
  <tbody>
    <tr>
      <td>Username</td>
      <td>xxxxxxxxxxxxx</td>
    </tr>
    <tr>
      <td>Password</td>
      <td>yyyyyyyyyyyyyy</td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <td colspan = "2">
        <span style = "display:block; text-align:center">
          <input type = "submit" name = "submit" value = "Submit" />
        </span>
      </td>
    </tr>
  </tfoot>
</table>
```

Listing 4.9: Table example

Listing 4.10 shows how a table is transformed with less tags and easier to read.

```
<h:panelGrid column="2" id="example" border="1" cellpadding="10"
    cellspacing="1">
  <f:facet name="thead">
    <h:outputLabel>Login</h:outputLabel>
  </f:facet>
  <h:outputLabel>Username</h:outputLabel>
  <h:outputLabel>xxxxxxxxxxxxx</h:outputLabel>
  <h:outputLabel>Password</h:outputLabel>
  <h:outputLabel>yyyyyyyyyyyyyy</h:outputLabel>
  <f:facet name="tfoot">
    <h:outputText style="display:block; text-align:center">
      <h:commandButton name="submit" value="Submit" />
  </h:outputText>
</f:facet>
```

Listing 4.10: Table result

4.2.6. If/Choose

Listing 4.11 has an example on how an if/else instruction would look in Java.

```
<% if (num > 0.95) { %>
  <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
<% } else { %>
  <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
<% } %>
```

Listing 4.11: if/else

Listing 4.12 shows how the transformation looks, with `<c:choose>`, `<c:when>` and `<c:otherwise>` tags.

```
<c:choose>
  <c:when test="#{num > 0.95}">
    <h2>You'll have a luck day!</h2>
    <p>(<%= num %>)</p>
  </c:when>
  <c:otherwise>
    <h2>Well, life goes on ... </h2>
    <p>(<%= num %>)</p>
  </c:otherwise>
</c:choose>
```

Listing 4.12: `<c:choose>`, `<c:when>` and `<c:otherwise>` example

4.2.7. foreach

Listing 4.13 has an example of how a table could be created in a JSP or pure Java environment, where all the rows are defined, all columns can be seen explicitly and there is a *for loop*.

```
<table border="1">
  <tr>
    <td>Festival Name:</td>
    <td>Location:</td>
    <td>Start Date:</td>
    <td>End Date:</td>
    <td>URL:</td>
```

```
  </tr>
  <% for(int i = 0; i < allFestivals.size(); i += 1) { %>
    <tr>
      <td>${allFestivals.get(i).getFestivalName()}</td>
      <td>${allFestivals.get(i).getLocation()}</td>
      <td>${allFestivals.get(i).getStartDate()}</td>
      <td>${allFestivals.get(i).getEndDate()}</td>
      <td>${allFestivals.get(i).getURL()}</td>
    </tr>
  <% } %>
</table>
```

Listing 4.13: Table with foreach example java

Listing 4.14 shows a `h:panelGrid` instead of a table, with the attribute *column* that has been calculated so there are no longer `trs` or `tds`. Now there are `h:outputLabel` tags that contain the text. There is a `<c:foreach>` instead of the *for loop*, so now it does not involve an index. The variable in the loop has a suffix *_elem* which substitutes the `.get(i)`, and the methods have a `get` word preceding it. These lines should be adapted according to what the variables in the Model are.

```
<h:panelGrid column="5" border="1">
<h:outputLabel>Festival Name:</h:outputLabel>
<h:outputLabel>Location:</h:outputLabel>
<h:outputLabel>Start Date:</h:outputLabel>
<h:outputLabel>End Date:</h:outputLabel>
<h:outputLabel>URL:</h:outputLabel>
  <c:forEach items="#{allFestivals}" var="allFestivals_elem">
    <h:outputLabel>${allFestivals_elem.getFestivalName}</h:outputLabel>
    <h:outputLabel>${allFestivals_elem.getLocation}</h:outputLabel>
    <h:outputLabel>${allFestivals_elem.getStartDate}</h:outputLabel>
    <h:outputLabel>${allFestivals_elem.getEndDate}</h:outputLabel>
    <h:outputLabel>${allFestivals_elem.getURL}</h:outputLabel>
  </c:forEach>
</h:panelGrid>
```

Listing 4.14: Table with foreach result

4.2.8. Google Skeleton

The code in listing 4.15 has the skeleton of the Google's home page, and its transformation into JSF can be seen in listing 4.16

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="css/style.css">
    <title>Google</title>
  </head>
  <body>
    <header>
      <ul>
        <li>Gmail</li>
        <li>Im genes</li>
        <li><img src="img/cuadros.png" /></li>
        <li><img src="img/campana.svg" /></li>
        <li><img src="img/perfil.jpg" /></li>
      </ul>
    </header>
    <main>
```

```
17      <section>
18        <img id="google" src="img/Google.jpg">
19      </section>
20      <section>
21          <input id="search" class="field" type="text" name="name"
                autofocus="true">
22          <img src="img/google-microfono.jpg" width=25 />
23      </section>
24      <section>
25        <button onclick="redirect()">Buscar en Google</button>
26        <button>Me siento con suerte</button>
27      </section>
28    </main>
29    <footer>
30      <ul>
31        <li>Publicidad</li>
32        <li>Negocios</li>
33        <li>Acerca de</li>
34      </ul>
35      <ul>
36        <li>Privacidad</li>
37        <li>Condiciones</li>
38        <li>Preferencias</li>
39        <li>Utilizar Google.com</li>
40      </ul>
41    </footer>
42  </body>
43 </html>
```

Listing 4.15: googleSkeleton.html

```
1  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://xmlns.jcp.org/
       jsf/html">
2  <h:head>
3    <h:outputStylesheet library="css" name="css/style.css">
4    <title>Google</title>
5  </h:head>
6  <h:body>
7  <h:header>
8     <ul>
9      <li>Gmail</li>
10     <li>Im?genes</li>
11     <li><h:graphicImage value="img/cuadros.png"/></li>
12     <li><h:graphicImage value="img/campana.svg"/></li>
13     <li><h:graphicImage value="img/perfil.jpg"/></li>
14    </ul>
15 </h:header>
16   <main>
17    <section>
18     <h:graphicImage id="google" value="img/Google.jpg"/>
19    </section>
20    <section>
21     <h:inputText id="search" styleClass="field" name="name" autofocus="true
          " />
22     <h:graphicImage width="25" value="img/google-microfono.jpg"/>
23    </section>
24    <section> <h:commandButton onclick="redirect()" value="Buscar en Google"
          />
25  <h:commandButton value="Me siento con suerte" />
26    </section>
27   </main>
```

```
28   <footer>
29    <ul>
30     <li>Publicidad</li>
31     <li>Negocios</li>
32     <li>Acerca de</li>
33    </ul>
34    <ul>
35     <li>Privacidad</li>
36     <li>Condiciones</li>
37     <li>Preferencias</li>
38     <li>Utilizar Google.com</li>
39    </ul>
40   </footer>
41   <h:outputScript library="js" name="" />
42 </h:body>
43 </html>
```

Listing 4.16: googleSkeleton.xhtml

Listing 4.17 is an example of the warning notes that are generated along the transformation.

```
1  ================================================================================
2  Warning notes and recommendations for the file index
3  ================================================================================
4  Check the tabulations and linebreakes for better readability
5
6  -- Link Tag Notes --
7  Check that there are no link tags in the file
8  Check that there are no empty attributes in the h:outputStylesheet tag
9  Check that both library and name attributes have coherent values
10
11 -- Script Tag Notes --
12 Check that there are no script tags in the file
13 Check that there are no empty attributes in the h:outputScript tag
14 Check that both library and name attributes have coherent values
15
16 -- Radio Tag Notes --
17 Check that there are no radio tags in the file
18 Check that there are no empty attributes in the f:selectItem tag
19 Check there are no label tags surrounding the f:selectItem tags
20 Check the itemValue attribute and the itemLabel value
21
22 -- Input Tag Notes --
23 Check that there are no input tags in the file
24 Check that there are no empty attributes in any of the following tags: h:
       inputText, h:commandButton, h:selectOneRadio, h:inputSecret, h:
       inputHidden, h:selectBooleanCheckbox
25
26 -- Option Tag Notes --
27 Check that there are no option tags in the file
28 Check that there are no empty attributes in the f:selectItem tag
29
30 -- A Tag Notes --
31 Check that there are no a tags in the file
32 Check that there are no empty attributes in the h:outputLink tag
33 Check the outcome attribute because it should contain the text in the link
34
35 -- Image Tag Notes --
36 Check that there are no img tags in the file
37 Check that there are no empty attributes in the h:graphicImage tag
38
```

```
39 -- Button Tag Notes --
40 Check that there are no button tags in the file
41 Check that there are no empty attributes in the h:commandButton tag
42
43 -- Table Tag Notes --
44 Check that there are no table, tbody, thead, tfoot, tr or td tags in the
       file
45 Check that there are no empty attributes in the h:panelGrid or f:facet tags
```

Listing 4.17: Notes example

Chapter 5

Conclusions and Future Work

Summary: This chapter presents the conclusions drawn during the development of the project as well as possible improvements that could be implemented to extend the functionality of the system or improve its efficiency.

5.1. Conclusions

The objective of the thesis set at the beginning, which was to create a software tool capable of migrating a web view file automatically from JSP to JSF, was achieved succesfully. The transformation from one file to another is clear, even though there are a few things that could improve and would have to be addressed in future versions of the project.

Before enrolling in the master I had worked with web technologies for a few years, so I chose to work on this project because it involved learning new web technologies that I had not worked with before but I could encounter in future jobs/projects, even though it did not involve creating create a web application per se. This project forced me to investigate and reinforce the knowledge I have about web technologies, even though it did not imply developing a web application.

Implementing this project included many challenges. Firstly to work again with Java, because in my case I had not done that since leaving college four years ago. Then I had to learn two technologies with which I had no experience with, compare them and create a tool that migrates from one technology to the other. Another challenge was also working with the Java packages, especially `Jsoup`, because of the challenges that emerged. The structure and purpose of the project forced to have a DOM String with the changes instead of making the transformations directly with `Jsoup`. It had been a while since I developed a program without using a Framework.

My previous experience working with web technologies along with the new knowledge and habits acquired in the master helped me get through this project despite what the COVID-19 circumstances are and the stress the situation provokes, so I think this is a good way to finish the master.

In summary, the balance is positive, especially because of the knowledge adquired during the development of the project.

5.2. Future Work

The future work to keep growing this project would imply to keep on making tests so that it can cover more cases, stress the system with more and more complex templates or even with real projects in order to tune the system and then get a more accurate transformation.

There are tags that have multiple HTML equivalences, so a better transformation could be to analyze and make sure which tag should be used in each case.

Other improvement would be better handling of tabs and spaces between tags and breaklines, so the resulting transformed file is more readable. There is also room for improvement on improving the warning notes and make them more precise with what the transformation encounters in the file or tries to do.

In a MVC Framework, this project only focuses on the view files, so a next step could be to work with servlet files in order to create a robust Controller file to address a fullfledged MVC transformation of the project and not just limit it to the view phase.

References

[1] Eric Normand. Why do we use web frameworks? https://lispcast.com/why-web-frameworks/, 2020 (Retrieved on April 1 2020).

[2] Kitty Gupta. What is the difference between jsp and jsf? https://www.freelancinggig.com/blog/2018/03/16/difference-jsp-jsf/, 2020 (Retrieved on April 1 2020).

[3] Paul Vasiliev. Five biggest challenges of software migration projects. https://www.syberry.com/company/blog/articles/streamlining-the-software-migration-process, 2020 (Retrieved on April 1 2020).

[4] Jay Wengrow. Is html a programming language? https://anyonecanlearntocode.com/blog_posts/is-html-a-programming-language, 2020 (Retrieved on April 2 2020).

[5] Krishna Eydat. Different versions of html. http://www.codefreetutorial.com/learn-html/76-different-versions-of-html, 2020 (Retrieved on March 28 2020).

[6] W3cschool. Html history. https://www.w3schools.in/html-tutorial/history/, 2020 (Retrieved on March 28 2020).

[7] Ferenc Almasi. 10 best practices for html. https://medium.com/swlh/10-best-practices-for-html-542fb923b93, 2020 (Retrieved on April 2 2020).

[8] William Craig. 20 html best practices you should follow. https://www.webfx.com/blog/web-design/20-html-best-practices-you-should-follow/, 2020 (Retrieved on April 2 2020).

[9] Michael Müller. *Practical JSF in Java EE 8*. Electronic version, 2018.

[10] Vangie Beal. Http - hypertext transfer protocol. https://www.webopedia.com/TERM/H/HTTP.html, 2020 (Retrieved on April 13 2020).

[11] Mozilla. Http request methods. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods, 2020 (Retrieved on April 13 2020).

[12] Cloudfare. What is https? https://www.cloudflare.com/learning/ssl/what-is-https/, 2020 (Retrieved on April 13 2020).

[13] Chip Securis. What is asymmetric encryption? read symmetric vs. asymmetric encryption diversity. https://cheapsslsecurity.com/blog/what-is-asymmetric-encryption-understand-with-simple-examples/, 2020 (Retrieved on April 29 2020).

[14] Karwan Jacksi. Development history of the world wide web. *International Journal of Scientific and Technology Research*, 2019.

[15] Cambridge Semantics. An introduction to the semantic web. https://www.youtube.com/watch?v=V6BR9DrmUQA, 2020 (Retrieved on April 6 2020).

[16] Pradeep Saran. From history of web application development. https://www.devsaran.com/blog/history-web-application-development, 2020 (Retrieved on April 2 2020).

[17] Jesse James Garrett. Ajax: A new approach to web applications. *Adaptive Path*, 2005.

[18] Metalog. The semantic web made easy. https://www.w3.org/RDF/Metalog/docs/sw-easy, 2020 (Retrieved on April 4 2020).

[19] Sam Richard. What are progressive web apps? https://web.dev/what-are-pwas/, 2020 (Retrieved on April 6 2020).

[20] Google Chrome Developers. Progressive web apps - pwa roadshow. https://www.youtube.com/watch?v=z2JgN6Ae-Bo, 2020 (Retrieved on April 6 2020).

[21] Pankaj Patel. Types of parsers in compiler design. https://www.geeksforgeeks.org/types-of-parsers-in-compiler-design/, 2020 (Retrieved on April 12 2020).

[22] Dale Janssen. What is a parser? https://www.techopedia.com/definition/3854/parser, 2020 (Retrieved on April 12 2020).

[23] Allforrous. Java (programming language). https://en.wikipedia.org/wiki/Java_(programming_language), 2020 (Retrieved on April 16 2020).

[24] Kartik Thakral. Introduction to java servlets. https://www.geeksforgeeks.org/introduction-java-servlets/, 2020 (Retrieved on April 16 2020).

[25] Neha Vaidya. Introduction to java servlets: Java servlets tutorial. https://www.edureka.co/blog/java-servlets, 2020 (Retrieved on April 16 2020).

[26] Giulio Zambon. *Introducing JSP and Tomcat. In Beginning JSP, JSF and Tomcat Java Web Development (pp. 1–18).* Electronic version, 2012.

[27] Giulio Zambon. *JSP Elements. In Beginning JSP, JSF and Tomcat Java Web Development (pp. 19–47).* Electronic version, 2012.

[28] Giulio Zambon. *JSP in Action. In Beginning JSP, JSF and Tomcat Java Web Development (pp. 78–119).* Electronic version, 2012.

[29] Chatanya Singh. Jsp forward action tag - jsp tutorial. https://beginnersbook.com/2013/11/jsp-forward-action-tag/, 2020 (Retrieved on April 15 2020).

[30] Chatanya Singh. Jsp include action tag - jsp tutorial. https://beginnersbook.com/2013/11/jsp-include-action-tag/, 2020 (Retrieved on April 15 2020).

[31] Chatanya Singh. jsp:usebean, jsp:setproperty and jsp:getproperty action tags. https://beginnersbook.com/2013/11/jsp-usebean-setproperty-getproperty-action-tags/, 2020 (Retrieved on April 15 2020).

[32] Giulio Zambon. *JavaServer Faces 2.2. In Beginning JSP, JSF and Tomcat (pp. 188–229).* Electronic version, 2012.

[33] Arjan Tijms Bauke Scholtz. *History. In The Definitive Guide to JSF in Java EE 8 (pp. 1–12).* Electronic version, 2018.

[34] Arjan Tijms Bauke Scholtz. *Conversion and Validation In The Definitive Guide to JSF in Java EE 8 (pp. 149–191).* Electronic version, 2018.

[35] Matjaz Jug. Migration from legacy systems. *Statistical Office of the European Union (EUROSTAT)*, 2013.

[36] Dan Woods. Learning from pivotal about migrating legacy applications. https://www.forbes.com/sites/danwoods/2018/02/22/learning-from-pivotal-about-migrating-legacy-applications/, 2020 (Retrieved on April 7 2020).

[37] David McAmis. Migrating legacy applications. https://www.techrepublic.com/article/migrating-legacy-applications/, 2020 (Retrieved on April 7 2020).

[38] Alexander Shvets. Chain of responsibility. https://refactoring.guru/design-patterns/chain-of-responsibility, 2014 (Retrieved on May 15 2020).

[39] Brett Van Porter. Welcome to apache maven. https://maven.apache.org/, 2020 (Retrieved June 8, 2020).

[40] Jonathan Knudsen. The java.io package. https://bioinfo2.ugr.es/OReillyReferenceLibrary/java/fclass/ch11_js.htm, 1997 (Retrieved on May 14 2020).

[41] Yidong Fang. jsoup java html parser, with the best of html5 dom methods and css selectors. https://github.com/fangyidong/json-simple, 2014 (Retrieved on May 14 2020).

[42] Jonathan Hedley. jsoup java html parser, with the best of html5 dom methods and css selectors. https://jsoup.org/, 1997 (Retrieved on May 14 2020).

[43] geeksforgeeks. Java.util package in java. https://www.geeksforgeeks.org/java-util-package-java/, 2014 (Retrieved on May 14 2020).

[44] The Apache Software Foundation. Commons cli. http://commons.apache.org/proper/commons-cli/, 2019 (Retrieved May 31, 2020).

[45] Arjan Tijms Bauke Scholtz. *The Definitive Guide to JSF in Java EE 8*. Electronic version, 2018.

[46] Michael Müller. *JavaServer Faces. In Practical JSF in Java EE 8 (pp. 35–48)*. Electronic version, 2018.

[47] Giulio Zambon. *Beginning Jsp, Jsf and Tomcat Java Web Development*. Electronic version, 2012.

[48] Oleg Uryutin. A brief history of web app. https://medium.com/@aplextor/a-brief-history-of-web-app-50d188f30d, 2020 (Retrieved on April 2 2020).

[49] Royce Moroch. Semantic web. https://en.wikipedia.org/wiki/Semantic_Web, 2020 (Retrieved on April 5 2020).